

DOCTOR OF PHILOSOPHY

Uncertainty analysis in the Model Web

Richard Jones

2014

Aston University

Some pages of this thesis may have been removed for copyright restrictions.

If you have discovered material in AURA which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our [Takedown Policy](#) and [contact the service](#) immediately

Uncertainty analysis in the Model Web

RICHARD MATTHEW JONES

Doctor Of Philosophy



– ASTON UNIVERSITY –

May 2013

© Richard Matthew Jones, 2013. Richard Matthew Jones asserts his moral right to be identified as the author of this thesis. This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without proper acknowledgement.

ASTON UNIVERSITY

Uncertainty analysis in the Model Web

RICHARD MATTHEW JONES

Doctor Of Philosophy, 2013

Thesis Summary

This thesis provides a set of tools for managing uncertainty in Web-based models and workflows. To support the use of these tools, this thesis firstly provides a framework for exposing models through Web services. An introduction to uncertainty management, Web service interfaces, and workflow standards and technologies is given, with a particular focus on the geospatial domain. An existing specification for exposing geospatial models and processes, the Web Processing Service (WPS), is critically reviewed. A processing service framework is presented as a solution to usability issues with the WPS standard. The framework implements support for Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and JavaScript Object Notation (JSON), allowing models to be consumed by a variety of tools and software. Strategies for communicating with models from Web service interfaces are discussed, demonstrating the difficulty of exposing existing models on the Web.

This thesis then reviews existing mechanisms for uncertainty management, with an emphasis on emulator methods for building efficient statistical surrogate models. A tool is developed to solve accessibility issues with such methods, by providing a Web-based user interface and backend to ease the process of building and integrating emulators. These tools, plus the processing service framework, are applied to a real case study as part of the UncertWeb project. The usability of the framework is proved with the implementation of a Web-based workflow for predicting future crop yields in the UK, also demonstrating the abilities of the tools for emulator building and integration. Future directions for the development of the tools are discussed.

Keywords: emulation, Model Web, interoperability, uncertainty management

Contents

1	Introduction	12
1.1	Uncertainty	13
1.2	Interoperability	13
1.3	Scientific contribution	14
1.3.1	Thesis aims and objectives	15
1.4	Outline of thesis	16
1.5	Related publications and open-source software	17
2	Literature Review	19
2.1	Foreword	20
2.2	Models	20
2.2.1	Uncertainty	21
2.2.2	Sensitivity Analysis	25
2.2.3	Emulation	28
2.3	The Model Web	33
2.4	Technology for Web services	35
2.4.1	Domain independent	36
2.4.2	Geospatial specific	42
2.4.3	Data encoding formats	49
2.5	Web-based workflows	53
2.5.1	Standards and software	54
2.5.2	Geospatial service integration	57
2.5.3	Uncertainty in workflows	60
2.6	Summary	60
3	Models as Web services	63
3.1	Foreword	64
3.2	Requirements for the Model Web	64
3.3	Web service interfaces	65
3.3.1	The WPS standard	65
3.3.2	Processing service framework	72
3.4	Model and interface integration	89
3.4.1	Execution patterns	90
3.4.2	Deployment challenges	92
3.4.3	Implementing communication	94
3.5	Summary	101
4	Emulation in the Model Web	105
4.1	Introduction to emulators	106
4.2	A tool for emulator building	109
4.2.1	Backend Application Programming Interface (API)	109
4.2.2	Frontend Web application	116

4.3	Using an emulator	123
4.3.1	Emulator representation	124
4.3.2	Emulators as Web services	125
4.4	Extensions for Sensitivity Analysis (SA) and validation	128
4.4.1	A tool for SA in the Model Web	128
4.4.2	Supporting additional validation scenarios	130
4.5	Limitations	132
4.6	Summary	134
5	Case study	137
5.1	Foreword	138
5.2	Introducing the scenario	138
5.2.1	Models	138
5.2.2	Workflow	141
5.3	Web architecture development	142
5.3.1	Exposing models and data	142
5.3.2	Building the workflow	148
5.4	Emulation of AquaCrop	151
5.5	Summary	155
6	Conclusions	157
6.1	Thesis summary	158
6.1.1	Development of a framework for exposing models on the Web	158
6.1.2	Development of a Model Web tool for emulation	159
6.1.3	Thesis aims and objectives	160
6.2	Directions for future research	162

List of Figures

2.1	Probability Distribution Function (PDF) of the average December air temperature in Cambridge, where the shaded area is the probability the temperature will exceed 5°C.	22
2.2	Calculated Sobol measures: mean and total effects.	27
2.3	Training an emulator for a single input simulator.	29
2.4	An overview of how the WPS proxy service proposed by Sancho-Jiménez et al. (2008) operates.	47
2.5	Composing a workflow in Taverna.	54
3.1	A simplified class diagram for the processing service framework.	73
3.2	The workflow scenario composed in Taverna with processes exposed using the framework.	87
3.3	Input and output matching in Eclipse Business Process Execution Language (BPEL) Designer between processes exposed using the framework.	88
3.4	Input and output matching in Eclipse BPEL Designer between processes exposed on a WPS.	89
3.5	Communicating with a model on a separate machine to the Web service.	93
3.6	A diagram of the main classes comprising the matlab-connector library.	96
4.1	The stages making up the emulator building and validation process.	107
4.2	Updating input descriptions in the simulator specification.	119
4.3	Fixing an inactive input from the screening results view.	120
4.4	Histogram of the design points for input with identifier 'Rainfall'.	121
4.5	Setting emulator training parameters.	121
4.6	Standard score plot for the 'SimpleSimulator' example emulator.	122
4.7	Histogram and QQ plot of mean residuals for the 'SimpleSimulator' example emulator.	123
4.8	Running the emulator for 'SimpleSimulator' using the upload service client.	127
4.9	Selecting the Sobol SA method and setting associated parameters.	129
4.10	Main and total effects calculated by the Sobol method.	130
4.11	Importing predicted sample values from a comma-separated values (CSV) file.	131
5.1	A basic workflow composition for the Food and Environment Research Agency (FERA) case study.	140
5.2	Comparing yield estimates over a 24 year period using the Greenland visualisation tool.	150
5.3	Assigning an input identifier to a column of uploaded design points.	152
5.4	AquaCrop simulator versus emulator mean and median plots.	153
5.5	Standard score plot for the trained AquaCrop emulator.	153

List of Tables

2.1	A list of frequently used Hypertext Transfer Protocol (HTTP) methods.	36
2.2	A list of commonly used BPEL elements.	55
3.1	A list of methods defined by the <code>AbstractProcess</code> class.	74
3.2	A list of methods defined by the <code>AbstractXMLEncoding</code> class.	79
3.3	Allowed and disallowed HTTP requests called from origin http://www.uncertweb.org	85
3.4	An overview of Web service technologies.	103
4.1	A list of inputs for the ‘SimpleSimulator’ model.	110
4.2	A list of outputs for the ‘SimpleSimulator’ model.	110
4.3	A list of operations provided by the emulation API.	112
4.4	Additional and modified API operations developed for SA and validation.	128
5.1	A comparison of simulator and emulator evaluation times.	155

Listings

2.1	An Extensible Markup Language (XML) document representing a Ford Ka. . . .	37
2.2	An XML schema defining a car, used to validate Listing 2.1.	38
2.3	The use of namespaces to prevent naming conflicts.	39
2.4	A SOAP envelope containing a fault.	40
2.5	Extract from a WPS <i>DescribeProcess</i> response, showing the description of an input.	44
2.6	Extract from a WSDL document, showing the XML schema for an input.	48
2.7	A Geography Markup Language (GML) document representing a point.	49
2.8	An Observations & Measurements (O&M) Measurement representing the area of a field.	51
2.9	A normal distribution encoded in Uncertainty Markup Language (UncertML) versions 1 and 2.	51
2.10	A geospatial point represented as a GeoJSON object.	52
2.11	A key-value pair (KVP) encoded process execution request.	58
3.1	A comparison of abstract and concrete descriptions in XML schema.	70
3.2	A description of an input with additional type annotation.	71
3.3	The <code>run</code> method for a simple polygon buffering process.	75
3.4	The generated schema for a buffer process XML schema.	77
3.5	A JSON encoded request for a buffering process.	80
3.6	Executing the polygon buffer process using code generated by Apache Axis.	83
3.7	Buffering a polygon in JavaScript using the JSON interface.	85
3.8	A JSON encoded matlab-connector request.	97
3.9	The JSON encoded matlab-connector response to Listing 3.8.	97
3.10	A simple R script annotated for deployment on a WPS.	100
4.1	A <code>GetProcessDescription</code> request and response for the <code>SimpleSimulator</code> process.	113
4.2	An <code>EvaluateProcess</code> request and response for the <code>SimpleSimulator</code> process.	115
4.3	The <code>Remotable</code> module included by classes using API functionality.	117
4.4	Using reflection to create dynamically named variables in the <code>RemoteController</code>	118
4.5	JSON representation of the trained emulator for ‘ <code>SimpleSimulator</code> ’.	124
5.1	An uncertain transition matrix generated by the Web-enabled Land Capability Classification System (LCCS) model.	143
5.2	Executing LANDcape Scale Functional Allocation of Crops Temporally and Spatially (LandSFACTS) programatically with the landsfacts-interface library.	145

List of acronyms

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BPEL	Business Process Execution Language
CORBA	Common Object Request Broker Architecture
CORS	cross-origin resource sharing
CRS	Coordinate Reference System
CSV	comma-separated values
CSW	Catalog Service for the Web
DOM	Document Object Model
FAO	Food and Agriculture Organization of the United Nations
FAST	Fourier Amplitude Sensitivity Test
FERA	Food and Environment Research Agency
GIS	Geographic Information System
GML	Geography Markup Language
GUI	graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JTS	JTS Topology Suite
JVM	Java Virtual Machine
KVP	key-value pair
LandSFACTS	LANDscape Scale Functional Allocation of Crops Temporally and Spatially
LCCS	Land Capability Classification System
LHS	Latin hypercube sampling
MIME	Multipurpose Internet Mail Extensions
MUCM	Managing Uncertainty in Complex Models
MVC	Model-View-Controller

NaN	not a number
NCA	National Character Area
NILU	Norwegian Institute for Air Research
OASIS	Organization for the Advancement of Structured Information Standards
OAT	one-factor-at-a-time
OCR	optical character recognition
ODE	Orchestration Director Engine
O&M	Observations & Measurements
OGC	Open Geospatial Consortium
OpenMI	Open Modelling Interface
OWS	Open Geospatial Consortium (OGC) Web Services
PDF	Probability Distribution Function
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSS	Really Simple Syndication
SA	Sensitivity Analysis
SCUFL	Simple Conceptual Unified Flow Language
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOS	Sensor Observation Service
STAS	Spatio-temporal Aggregation Service
SWG	Standards Working Group
TSV	tab-separated values
UA	Uncertainty Analysis
UDDI	Universal Description, Discovery and Integration
UI	user interface
UncertML	Uncertainty Markup Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
VM	virtual machine

W3C	World Wide Web Consortium
WADL	Web Application Description Language
WCS	Web Coverage Service
WebDAV	Web Distributed Authoring and Versioning
WFS	Web Feature Service
WMS	Web Mapping Service
WMTS	Web Map Tile Service
WPS	Web Processing Service
WSDL	Web Service Description Language
XML	Extensible Markup Language
Xvfb	X virtual framebuffer

Acknowledgements

I would like to thank everyone who has either helped me, motivated me, distracted me, or put up with my bad moods throughout the course of the PhD: Dan, Lucy, Matt, Remi, Christoph, Marc, Morgane, Becky, Jamie, Alice, John, Tom and Jai. An extra special thanks to Mum, Dad and Rebecca.

1

Introduction

CONTENTS

1.1	Uncertainty	13
1.2	Interoperability	13
1.3	Scientific contribution	14
	1.3.1 Thesis aims and objectives	15
1.4	Outline of thesis	16
1.5	Related publications and open-source software	17

1.1 Uncertainty

The work in this thesis is based around the concept of uncertainty in scientific models. Uncertainty is where a lack of knowledge prevents us from being able to exactly describe a state or outcome. While we are unable to describe these exactly, we can describe them with a set of possible states or outcomes, and use probability theory to quantify the chance of each state or outcome occurring (Papoulis, 1984). The simplest example of uncertainty and probability theory in practice is the flip of an unbiased coin. As the chance of either outcome occurring is equal, the probability of heads is $\frac{1}{2}$ and the probability of tails is also $\frac{1}{2}$.

Models have long been used in science where physical experimentation is impractical or even impossible (Sacks et al., 1989). As approximations of reality, all models are uncertain, as the real-world processes they observe must be simplified. Additionally, measurements which form inputs to the models will be subject to a degree of uncertainty. Without quantifying these uncertainties and subsequently propagating them through to the model output, it is impossible to make informed judgements about the results, judgements which may inform critical decision making processes. The issue is even more critical when we consider the propagation of uncertainty in a model workflow, where models can potentially be discovered automatically, and the output must include uncertainty contributions from every model.

The propagation of uncertainty, often referred to as Uncertainty Analysis (UA), can be considered part of the blanket term ‘uncertainty management’. This term also encompasses analysis techniques to quantify the contributions of uncertainty on model inputs to the overall output uncertainty (known as Sensitivity Analysis (SA) (Saltelli et al., 2000)), and methods to provide computationally efficient statistical model surrogates (emulation (Shahsavani and Grimvall, 2011)). While these techniques and methods are well-researched, their application remains limited.

1.2 Interoperability

Interoperability is the ability of diverse systems to exchange information, and can be provided through the adoption of standard data formats and communication protocols. The concept is critical in computer science. Consider, for example, if Hypertext Markup Language (HTML) had not been developed as a standard format for creating Web pages. Every site we visit would potentially have a page written in a different way, thus requiring browsers to perform the impossible task of understanding an unlimited number of formats.

In the context of scientific models, interoperability involves the use of standard interfaces for model evaluation and prescribed formats for representing inputs and outputs. Once interoperability is achieved, it is possible to build tools and complex workflows consisting of several models, without the need to consider the interfaces and data formats of individual models. These ideas are central to the Model Web — a concept where models and data sources can be deployed as Web services, thus creating a diverse collection of globally accessible components (Geller and Turner, 2007). This represents a significant shift from traditional desktop-based applications, where models must be installed and configured on every machine where the model will be evaluated.

The work in this thesis forms part of the UncertWeb project¹. UncertWeb aims to uncertainty-enable the Model Web by developing mechanisms for representing and transferring uncertainty, building tools for visualising uncertainty, and the integration of these with workflows. However, as the Model Web is not yet a mature concept, UncertWeb had to first deploy models and data sources on the Web before considering the uncertainty aspect. While the Model Web can potentially include models and data sources from several disciplines, the concept was originally coined in the geospatial domain. Therefore, with additional consideration given to the requirements of the UncertWeb case studies, the work in this thesis will focus on managing uncertainty in the context of the geospatial Model Web.

The list of UncertWeb partners includes the Food and Environment Research Agency (FERA), based in the UK. The primary involvement of FERA in the UncertWeb project was the proposed use of Model Web components to build a workflow to predict the effects of climate change on wheat yields in England, involving models to classify field capabilities, simulate field usage, and calculate crop yields.

1.3 Scientific contribution

This thesis looks at mechanisms for managing uncertainty, with a specific aim of providing them to the geospatial Model Web. The intention here is not to evaluate or develop methods for uncertainty management, rather to increase the applicability of existing methods through the adoption of Web services. The contributions of this thesis to the scientific community can be summarised as follows:

- The primary contribution of this work is a set of tools to support emulation in the Model Web. While emulation techniques are well-researched, their use is currently restricted to a

¹<http://www.uncertweb.org/>

limited number of software and libraries for mathematical computing environments. The tools include a backend Application Programming Interface (API), allowing developers to integrate emulation methods into their software, and a frontend graphical user interface (GUI) to provide a user with parameter adjustment and visualisations at each stage of the emulation process. Through integration with the Model Web, and utilising the interoperability provided, the tools can support a variety of models, without requiring the use of model-specific code.

- A secondary contribution of this research is the development of a framework for exposing processes and models on the Web. The processing service framework allows developers to deploy functionality using Simple Object Access Protocol (SOAP) and JavaScript Object Notation (JSON) technologies, without requiring knowledge of the standards themselves. Appropriate data encoding standards are automatically selected by the framework, further minimising the effort required by the process developer. Through adopting existing, widely-used standards, processes and models exposed using the framework can be executed by third-party libraries, workflow software, and Web applications.
- A further contribution is the development of mechanisms and libraries to enable programmatic model evaluation. Although Web service interfaces can enhance the interoperability and usability of models, the service implementation must communicate with the underlying model. A library developed for MATLAB enables functions to be executed from Java with ease, while other libraries were developed to provide the same benefits for two existing models, LANDcape Scale Functional Allocation of Crops Temporally and Spatially (LandSFACTS) and AquaCrop.

1.3.1 Thesis aims and objectives

The three objectives below outline the primary research aims of this thesis:

Objective 1 — Can the accessibility of emulation techniques be improved by Web services? This includes their adoption to provide tools for building emulators, and subsequently using those emulators for sensitivity and uncertainty analyses.

Objective 2 — In the context of the Model Web, what is the most interoperable and usable Web service interface for exposing models? Through interoperability, a single client implementation can potentially access an unlimited amount of models, and support integration with

other Web service workflow components.

Objective 3 — Can the Web service interface support the exposure of existing models, without substantial technical knowledge? For example, by providing communication bridges and automatic Web service process generation for popular modelling frameworks and software.

1.4 Outline of thesis

Chapter 1 is this introduction.

Chapter 2 discusses uncertainty, and specifically various methods for managing uncertainty in scientific models, which includes emulation. The concept of the Model Web is introduced, followed by a review of service interface and data standards aiming to provide interoperability to support Service Oriented Architectures (SOAs). Finally, approaches to composing and orchestrating model workflows on the Web are discussed.

Chapter 3 reviews the Web Processing Service (WPS) specification as an interface for exposing models on the Web. A critical analysis reveals the standard to lack in the level of interoperability and usability required to support the adoption of the Model Web. An alternative framework is designed and developed, aiming to solve the limitations of WPS. The usability of models exposed with the framework is evaluated, with a focus on integration with existing technologies and software. Finally, the challenges of exposing models on the Web are outlined, and potential approaches for Web service interface to model communication are discussed.

Chapter 4 introduces a set of tools to increase the accessibility of emulator methods in the Model Web. A backend API is developed, allowing language-independent access to emulator building functionality for Web-enabled models. A frontend is also developed, guiding a user through the stages of emulation with a series of parameter adjustment forms and visualisations. These tools were further developed to support additional uncertainty management mechanisms, SA and validation. Finally, the limitations of the developed tools are discussed.

Chapter 5 details the implementation of the FERA case study to predict future crop yields, with a focus on evaluating the usability of the processing service framework and emulation tools. The exposure of each model in the case study is detailed, demonstrating the effort required in contributing to the Model Web. A workflow containing these models is orchestrated

using JavaScript, proving the usability of the framework. Finally, the emulation tools are employed to build and deploy an emulator, greatly reducing the time to perform uncertainty analysis on a yield calculation model.

Chapter 6 summarises the work presented in this thesis and identifies future directions of research.

1.5 Related publications and open-source software

The work presented in this thesis is original and not been published elsewhere. However, parts of the work have been presented in the following conferences and papers:

- A review of mechanisms for orchestrating uncertainty-enabled model workflows was presented at the Workshop on Workflows for Earth Observation Systems in 2010 (oral presentation).
- Preliminary work on handling uncertainty in Web-enabled model workflows was presented at Spatial Accuracy in 2010 (oral presentation) (Jones et al., 2010).
- A discussion on the reality of deploying models in SOAs was presented at European Geosciences Union in 2011 (poster presentation).
- The tool to support emulation in the Model Web was presented at European Geosciences Union in 2012 (oral presentation).
- The design, development and use of the processing service framework was published in Transactions in GIS (Jones et al., 2012).
- A discussion on the practical implementation issues of emulator methods was published in Environmental Modelling & Software (Bastin et al., 2013).

The research undertaken by this thesis has also resulted in the development of the following open-source software:

- A SOAP/WSDL and JSON framework for processing services², plus a transactional extension for deploying emulators³.

²<https://github.com/itszootime/ps-framework>

³<https://github.com/itszootime/ps-emulatorized>

- An API and Web frontend for building emulators^{4,5}.
- A library for remotely executing functions on MATLAB instances⁶.
- A Java interface for a local LandSFACTS instance⁷.
- A Java interface for a local or remote AquaCrop instance⁸.

⁴<https://github.com/itszootime/emulatorization-web>

⁵<https://github.com/itszootime/emulatorization-api>

⁶<https://github.com/itszootime/matlab-connector>

⁷<https://github.com/itszootime/landsfacts-interface>

⁸<https://github.com/itszootime/aquacrop-interface>

2

Literature Review

CONTENTS

2.1	Foreword	20
2.2	Models	20
2.2.1	Uncertainty	21
2.2.2	Sensitivity Analysis	25
2.2.3	Emulation	28
2.3	The Model Web	33
2.4	Technology for Web services	35
2.4.1	Domain independent	36
2.4.2	Geospatial specific	42
2.4.3	Data encoding formats	49
2.5	Web-based workflows	53
2.5.1	Standards and software	54
2.5.2	Geospatial service integration	57
2.5.3	Uncertainty in workflows	60
2.6	Summary	60

2.1 Foreword

The underlying focus of this chapter is scientific models (Section 2.2) — their integration with the Web (Section 2.3), the technology to realise this integration (Section 2.4), and the composition of multiple models to perform complex analysis (Section 2.5).

Section 2.2 introduces models, and their use for scientific research purposes. The importance of uncertainty in modelling is discussed in Section 2.2.1, including how uncertainty is represented and propagated from model input to output. Sections 2.2.2 and 2.2.3 respectively tackle two questions introduced by the notion of uncertainty in modelling — how to decide which inputs to prioritise when gathering accurate measurements, and how to speed up uncertainty analysis techniques. Deploying models as services on the Web brings many opportunities for sharing and reuse. The concept supporting this, known as the Model Web, is introduced in Section 2.3.

The technology which could support the realisation of the Model Web, and the challenges associated with a practical implementation, are discussed in Section 2.4. This discussion covers some service standards that are domain independent (Section 2.4.1) and some that are specific to geospatial contexts (Section 2.4.2), and provides a comparison of several approaches. Data encoding standards are vital to ensure that clients and services from different organisations can communicate, and Section 2.4.3 details such standards for observations, geospatial data, and uncertainty encoding.

Complex analysis can be performed by building workflows composed of smaller, less-complex models. Section 2.5 discusses the technology to support workflow composition on the Web, including software and standards (Section 2.5.1) and how this technology is applied in the geospatial domain (Section 2.5.2).

2.2 Models

In scientific research, the collection of observations through physical experimentation alone can be time consuming, expensive, and even impossible in cases such as weather forecasting (Sacks et al., 1989). The scale of the issue grows as the complexity of the observed real-world systems and processes increases. As a result, many experimenters have adopted mathematical or computational models. A model is an approximate representation of a system or process, developed to allow an investigator to explore properties of interest. The output of a model represents properties of the system, given that appropriate input factors are provided. For example: a field use sim-

ulator might take crop history and climate data as inputs. The output would include time series of predicted crop types in each field. A mathematical model aims to characterise a system with a series of equations, input factors and parameters, in practice yielding results which would be extremely impractical for humans to produce through manual calculation. The computer program or machine that implements a model is commonly referred to as a simulator, and this thesis adopts this definition.

2.2.1 Uncertainty

Uncertainty is a state of having doubt or limited knowledge, which makes it impossible to describe an existing state, a future outcome, or a set of outcomes, in a deterministic manner. Uncertainty affects our lives in many ways — consider the following questions: will it rain today? Will ‘According to Pete’ win the Grand National this year? Will having a PhD substantially improve my career prospects? All are subject to uncertainty, and therefore impossible to answer definitely. In science, the measurement of phenomena and the accuracy of these measurements is a primary source of uncertainty. When scientific models are used, uncertainty can also manifest in the output produced, either as a by-product of uncertainty in the input, as a result of underlying variability in the model itself, or because there is model error or model discrepancy. As approximations of reality, all models are inherently wrong, meaning that even deterministic models should be considered uncertain. If uncertain information will form the basis for a decision, it is important that the uncertainty is recognised and accounted for.

Representing uncertainty

Uncertainty can be represented in a number of different ways, either qualitatively and quantitatively. The representation used depends on the eventual uses for the uncertainty. In everyday conversation, the question “is it going to rain today?” could be answered with “I am fairly sure it will” — a simple qualitative description of our uncertainty. As probability theory may be considered excessive for everyday conversation, such representation is suitable for this situation, but clearly inadequate for thorough statistical analysis.

Probability theory is a branch of mathematics that uses probabilities to quantify uncertainty, and is overwhelmingly argued to be the most appropriate representation of uncertainty (O’Hagan, 2011). A common example to demonstrate probability theory is the roll of a 6-sided die. Assuming the die is fair, the probability of any number appearing is $\frac{1}{6}$, and the probability of an even number

appearing is $\frac{1}{2}$, or 50%. In betting on either of these outcomes, the probabilities tell us that the chance of an even number appearing is higher than the chance of a specific number, thus allowing us to make an informed decision. Although this is clearly a trivial example, probability theory can be applied in more complex scenarios, providing the decision making process with more information.

The set of possible outcomes of an experiment is known as the sample space, and a probability distribution identifies the probabilities for the occurrence of each individual outcome in the sample space. The roll of a die example has a finite sample space containing six possible outcomes, and is therefore described by a discrete probability distribution. As physical variables such as time, distance, and air temperature have an infinite number of possible outcomes (i.e. measured values), we must instead use a continuous probability distribution. The Probability Distribution Function (PDF) is a curve beneath which the area describes the total of all probabilities in a continuous probability distribution. We consider the area under the curve when calculating probabilities; for example in Figure 2.1, the probability that a value lies between two bounds is related to a section of the area.

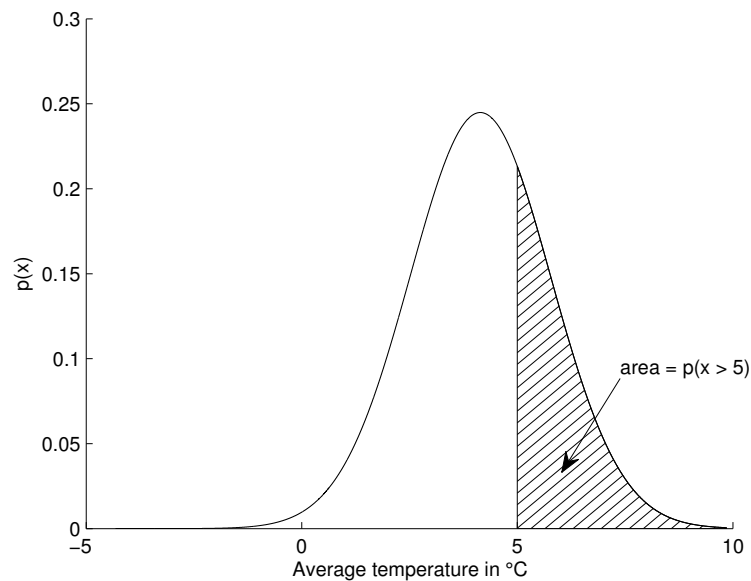


Figure 2.1: PDF of the average December air temperature in Cambridge, where the shaded area is the probability the temperature will exceed 5°C.

The notion of a random variable (a variable whose value varies as a result of chance or randomness) is fundamental to probability theory, and we consider every uncertain observation or data derived from observations as a random variable. A PDF describing the probabilities that the variable will take a given value is typically characterised by a functional form and a set of parameters. For example, in the case of the Gaussian distribution, there are two parameters: location (μ)

and scale (σ^2). These parameters relate to statistics describing the distribution — the location is the mean, or expected value, and the scale is the variance, a measure of how much the values of the random variable vary from the expected value.

Fuzzy logic is an alternative to probability theory, based on set theory. Traditional set theory treats membership as a binary value, 0 if an element is not in the set, 1 if it is. In contrast, fuzzy set theory assigns a truth value to membership — a value ranging between 0 and 1, describing the degree to which the element is a member of the set. Whilst a completely probabilistic approach is adopted in this thesis, it is important to recognise that other quantitative representations of uncertainty are available.

Uncertainty in modelling

Due to the practical benefits they provide, models are increasingly being used for decision making, and the use of complete information as inputs will improve decision making. Decision making often involves a cost if an action is taken, or a loss is incurred by taking no action. Uncertainty is therefore critical if decisions will be made based, partially or wholly, on the output of a model. A failure to account for and quantify model output uncertainty could result in the decision maker being provided with a misleading result, if it is assumed that the output is representative of all the information available. In reality, significant uncertainty may exist in model inputs and the model structure itself, and both may contribute to uncertainty in the model output.

Model inputs are often observations of the real world, used either directly from a measurement, or in the form of data derived from observations, and are subject to many sources of uncertainty. Measurement uncertainty can be a result of defects or design problems on the instrument used to measure. A sensor manufacturer may supply a sensor bias estimate derived from lab experiments, with which measurements can be corrected. However, if no estimate is provided, the unknown bias must be treated as uncertainty. Observations are also subject to representation uncertainty (an inability to describe the exact property of interest) as they may be influenced by surroundings and natural variability making it impossible to accurately measure a value at a single point in time. These various sources of error mean the true value of an input is rarely known.

Observation uncertainties are often predefined based on sensor characteristics, and if they are not, they can be estimated through physical and lab-based calibration methods. Uncertainty can also be estimated after a measurement has been made, using a set of quality-controlled observations as reference values for the phenomenon of interest. By comparing the new observations to

the reference values, an overall uncertainty estimate can be calculated.

Model structure uncertainty is the error contributed by the modelling process itself. As an approximation of reality, a model is inherently subject to uncertainty. The real-world processes we observe, whether they be physical, chemical, biological or human, must be simplified for inclusion in the model (Refsgaard et al., 2006). These processes may involve space and time, two fields which each cover an infinite domain. A computational model must map these to finite discrete representations, typically by projecting onto a grid for space, or sampling regular values for time. The quantisation from infinite continuous fields to finite discrete variables causes information to be lost, hence introducing uncertainty. A model may contain inputs which cannot be directly observed, and which therefore remain as fixed parameter values in the model. However, these values are often determined by means of observation or experimentation, and can therefore contribute to uncertainty in the model output. Where observation or experimentation is impossible, expert elicitation techniques can be employed. A group of experts, knowledgeable in the domain of the problem and phenomenon under study, all individually submit judgements to characterise the variable. When completed by all experts, the elicited knowledge is combined to create a pooled PDF describing the parameter and the uncertainty around it.

While it is important to quantify the inherent limitations of the modelling approach, understanding the relationship between the model and the real-world process behaviour is a challenge (Goldstein and Rougier, 2009), and an open research problem. As a result, the work in this thesis focuses on only model input uncertainty, and does not consider uncertainty introduced by model parameters or structure.

Analysis of uncertainty

Uncertainty Analysis (UA), often referred to as error propagation (Heuvelink, 1998), aims to quantify the uncertainty in a model output as a result of uncertainty in the input. When the true value of a model input is unknown, the goal of UA is to quantify information lost as a result (Oakley and O'Hagan, 2002). The results of UA can help to make informed decisions when faced with unknown inputs, or, if the resulting output uncertainty is high, conclude that the output is an unsuitable basis for decision making.

UA is typically achieved using Monte Carlo methods, where random sampling is performed to compute results. Once the input distributions are sampled, the model is evaluated for each sample. Provided that the sample is large enough, we can subsequently make accurate judgements about

the output distribution (Oakley and O'Hagan, 2002). Limitations of UA are clear when the model under evaluation is complex, or has a large number of inputs. With a relatively small sample size of 1000, and a model with an evaluation time of 1 minute, UA will take over 16 hours. When dealing with a large number of inputs, we must increase the sample size to ensure that representative combinations of different input values are included in the sample, thereby increasing the total time demands for UA. The Monte Carlo approach has additional overhead, in that sampling must be performed, and a mechanism is required to control multiple model evaluations.

2.2.2 Sensitivity Analysis

As discussed in Section 2.2.1, model inputs are subject to many sources of uncertainty, limiting our confidence in the output of a model. While UA quantifies the overall output uncertainty as a result of uncertainties in the input (Saltelli et al., 2008), it does not explore the relationships between model input and output. Sensitivity Analysis (SA) is the study of how different sources of variation contribute, quantitatively or qualitatively, to variation in the output of a model (Saltelli et al., 2000). It studies the behaviour of a model corresponding to the information flowing in and out, and the relationships between those information flows. As a result of understanding model behaviour, and more specifically understanding how a model output responds to changes in the inputs, SA aims to increase the confidence in predictions made by a model.

SA is of use in decision making, communication between modellers and decision makers, understanding or quantification of a system, and model development (Pannell, 1997). Assessing model behaviour can help focus time and expense on gathering accurate measurements for inputs that are more sensitive or important than others. SA can provide a means for model diagnostics, ensuring that inputs intended to be influential are demonstrating the desired effect on the output, and thus helping understand the relationships between model inputs, and variables in the system or process of interest. When the contributions of different inputs are quantified, model simplification is possible — unimportant inputs can be removed or fixed. Acquiring understanding of a system using SA may be even more critical if the model was developed by a third-party, provides minimal documentation, or hides implementation detail. The use of SA, in combination with UA, is considered by some to be a critical requirement in model building and scientific analysis (Hamby, 1994).

SA can be performed with a number of different methods, each of which has strengths and weaknesses, and is more or less suitable for models exhibiting a certain type of behaviour. The

main distinguishing factor between techniques is their handling of input factors, which may be local or global.

Local SA is performed by computing measures of how the output functions change in respect to the input factors. These measures, known as partial derivatives, are calculated numerically by perturbing input variables marginally from their baseline values (Rabitz, 1989; Turányi, 1990). The amount of variation is not derived from our knowledge of the factor, and is typically kept constant between all inputs. Local SA is an example of an one-factor-at-a-time (OAT) approach, where one factor is varied whilst all other factors are held constant (Saltelli et al., 2000). The OAT approach does not consider the effect of varying two factors simultaneously, and thus does not capture interactions between inputs (Czitrom, 1999).

As they only focus on the local impact of input factors on the model output, it is extremely unlikely that local SA methods will consider the plausible input ranges fully. With non-linear models, output varies differently across the space of the input factors. With a limited number of permutations, the perturbed values may not cover the space where a factor has significant impact on the model. The modeller or decision maker could potentially be given misleading results, which quantify behaviour that only exists when considering limited or unrealistic input ranges. However, as local SA methods are easily applied and require fewer model evaluations, they can still be of use for ranking key inputs in linear models (Frey and Patil, 2002).

In practice, understanding the consequences of input uncertainty requires more than small changes to baseline input values (Oakley and O'Hagan, 2004). Global SA considers output variation over a defined range of input space, making it more informative for non-linear models (Homma and Saltelli, 1996) and for cases where uncertainty of factors varies in orders of magnitude (Cukier et al., 1973). Our knowledge, or lack of knowledge, about a continuous input is sometimes represented by a PDF, or a set of value ranges. Performing a global SA involves varying the input factors in accordance with the PDFs or ranges and repeatedly evaluating the model for each combination of sampled values. The steps for performing a global SA are:

1. Elicit ranges for each input factor. This can be extended to non-uniform input ranges, where PDFs are used instead.
2. Create the experiment design — a set of points to cover the input space. If a PDF was specified, sample from that.
3. Evaluate the model at the points in the design.

4. Calculate the influences of each input factor on the output.

The first sensitivity measure, known as the ‘main effect’, is how much a factor contributes to the output variance. Many global variance-based SA methods are capable of computing the main effect. As it is unlikely that factors will be independent, it is also important to consider the interactions and influences of all factors (Chan et al., 1997). In addition to the main effect, the Sobol method calculates the ‘total effect’ for each factor, which includes the main effect plus variance contributions from all combinations with other factors (Sobol, 1993). Although the classical Fourier Amplitude Sensitivity Test (FAST) method does not compute total effects (Saltelli and Bolado, 1998), it was later extended to quantify the contributions of interaction between inputs (Saltelli et al., 1999). Figure 2.2 shows a plot of main and total effect measures computed by the Sobol method for a model with 8 input factors, where X_1 and X_2 are shown to have the greatest effect on the model output. Note that these measures are indices, and therefore represent a proportion of the total variance.

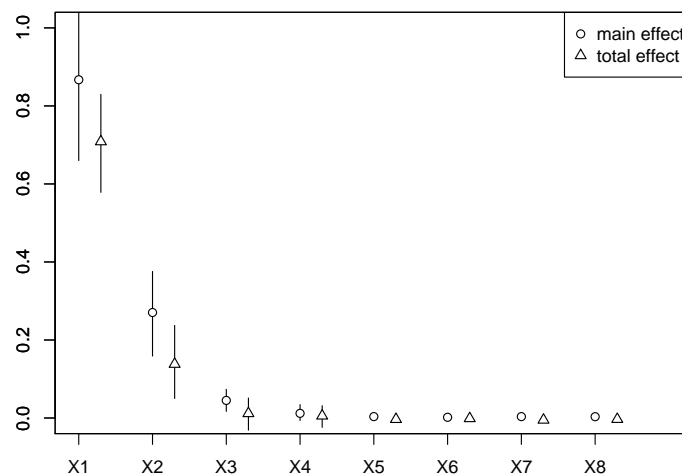


Figure 2.2: Calculated Sobol measures: mean and total effects.

Sobol and FAST make no assumptions about the model structure (Chan et al., 1997; Saltelli et al., 2000), and can therefore cope with non-linear (and non-monotonic) models. Saltelli and Bolado (1998) found that measures calculated by the two methods are equal in most test cases, with FAST being computationally cheaper. However, in that comparison only the classical FAST method was used, giving Sobol the advantage of computing total effects. Other variance-based methods are available, such as importance measures (Iman and Hora, 1990) and correlation ratios (Krzakacz, 1990; McKay, 1995). It has been shown that these methods are less efficient than Sobol and FAST (Saltelli et al., 2000). As a further drawback, they only calculate the main effects,

and do not account for the interaction effects.

As global SA considers the whole range of input uncertainty, it typically requires a greater number of model runs than a local method, therefore analysis is slower compared to local methods. The elicited PDFs or ranges will often vary between uses of the model, for example: parameter ranges for a crop yield model will be different if the crop of interest is wheat or potatoes. It is therefore usually impossible to provide a universal confidence assessment for a model.

Screening methods are a form of qualitative SA, and can be based on local or global techniques. Instead of quantifying the relative importance of input factors, screening simply ranks the order of importance (Saltelli et al., 2004). The benefit of screening is provided by the efficient designs, where the size is linear to the number of model factors. Such designs therefore require fewer model evaluations in comparison to quantitative methods. These characteristics mean that it is well suited to initial experiments, model selection, and identification of inputs that have a negligible effect on the output.

The screening method with the greatest applicability is that proposed by Morris (1991). Two measures are computed for each factor: μ , the overall effect the factor has on the output, and σ , the non-linear and interaction effects. In the original method, design points from the input space are selected at random, potentially leaving areas uncovered by the design. This issue was later resolved by Campolongo et al. (2007), who propose an enhanced Morris method to cover the design space by maximising the distance between points.

SA methods, including Sobol, FAST, and Morris have been implemented in standalone tools and libraries for popular numerical and statistical computing environments. An unexplored area of research is the usability of these tools and libraries, and how they integrate with other model components, especially those on the Web. If a modeller currently wishes to perform SA, they may be required to study the vast array of mathematical and theoretical techniques and reviews (Pannell, 1997), presenting a significantly high barrier to use.

2.2.3 Emulation

Monte Carlo, Sobol, and FAST are computationally expensive due to the requirement for a large number of model runs. For a complex model with a relatively long evaluation time, performing these analyses can be impractical. As the number of model inputs increases, a greater number of model runs is required to ensure full coverage of the input space. Performing a comprehensive analysis may require millions of runs, and even for a model with an evaluation time of one second,

this will take days to complete. The number of runs required can be reduced by optimised experimental designs, such as those proposed by Morris (1991), which aim to cover the largest area of space possible with the fewest design points. However, such methods provide qualitative instead of quantitative analysis, and therefore cannot be used to apportion variance to each input.

In the context of modelling, an emulator is a statistical approximation of a simulator (Kennedy and O'Hagan, 2001), and provides an alternative strategy to efficient experimental designs. We consider the simulator as a function $f(\cdot)$, which produces an output y given input vector x , such that $y = f(x)$. If an approximation $\hat{f}(\cdot)$ is accurate and produces measures close enough to those that would have been calculated using the simulator $f(\cdot)$, we can instead use this approximation in UA (Oakley and O'Hagan, 2002) and SA (Oakley and O'Hagan, 2004). Instead of providing a single value $\hat{f}(x)$ as an approximation for $f(x)$, the statistical nature of an emulator provides the entire probability distribution. The output for $\hat{f}(x)$ can thus be considered the mean of that distribution, and the accuracy of the prediction is described by a distribution around that mean.

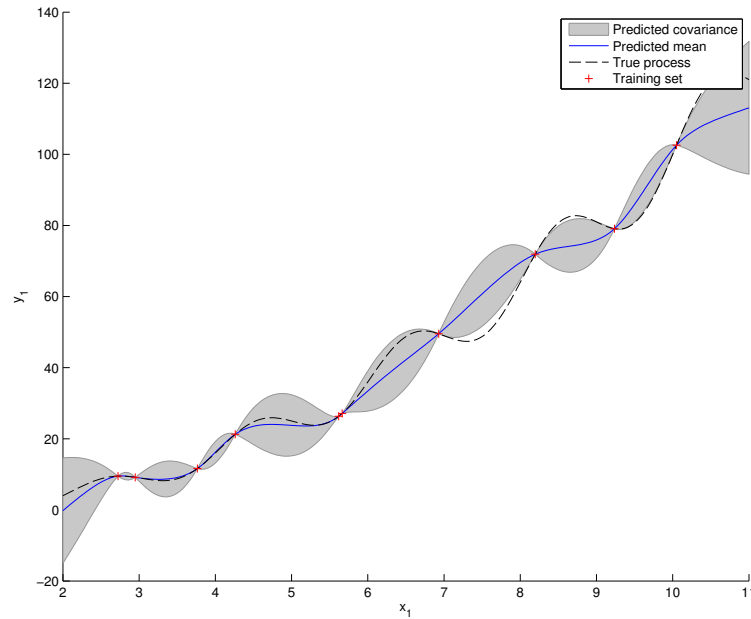


Figure 2.3: Training an emulator for a single input simulator.

A number of runs on the model are performed to create the training set — x and y values forming the basis for $\hat{f}(\cdot)$ estimations. At a point x on the training set, $\hat{f}(x)$ should return the associated y value from the set with no uncertainty, as the true output of the simulator is known. Otherwise, interpolation of the training data should be used, with the distribution around the returned mean representing the uncertainty as a result of interpolation (Currin et al., 1991). Figure 2.3 shows an example plot of an emulator trained for a simple model with a single input x_1 and output y_1 . The shaded grey area represents the uncertainty of the predictions made by the emulator. We observe a

growth in uncertainty when the input value x_1 is at greater distance from points in the training set.

Building an emulator is a complicated and iterative process, ideally involving input from model owners, builders, and experts in the problem domain. The main steps for building an emulator are:

1. Elicit ranges for each input factor — expert beliefs about the input values are necessary as we only want to train the emulator over plausible ranges.
2. Create the training set — generate a design of points to cover the input space, typically with a sampling method such as Latin Hypercube (Santner et al., 2003). The simulator is then evaluated at these points, resulting in a training set of x - y pairs.
3. Train the emulator — by selecting functions for approximation and their associated parameters.
4. Validate the emulator — essential to ensure that the emulator accurately represents the simulator (Bastos and O’Hagan, 2009). If validation results are unsatisfactory, training can be repeated with different parameters, or a larger design to improve emulator coverage.
5. Use the emulator — as a surrogate for the simulator in UA and SA.

Gaussian processes can be used to describe the behaviour of an unknown mathematical function (O’Hagan and Kingman, 1978), and as we consider the model simulator $f(\cdot)$ to be a mathematical function, they are commonly employed by emulators. In an emulator, a Gaussian process with mean and covariance functions models the simulator output and the uncertainty about it (Bastos and O’Hagan, 2009). The emulator builder must select appropriate mean and covariance functions to describe the simulator output, which will vary depending on characteristics of the underlying model.

The emulator training process has parameters, some of which are specified by the emulator builder, and others which are estimated. One estimated parameter, smoothness, relates to the degree to which a function is affected by small changes in the inputs (O’Hagan, 2006). If a function is greatly affected by small changes in the inputs, it is considered rough, and will require a larger training design. Estimating correct smoothness values is critical, as it affects the confidence of the emulator. If a smoothness value is high, but the function is actually rough, predictions made by the emulator will be over-confident — they have low uncertainty even though they are far away from the simulator output. If a smoothness value is low, but the function is actually smooth, predictions will be under-confident — they have high uncertainty when making accurate predictions.

Creating an appropriate training design is critical to the emulator building process, as the Gaussian process will be trained to fit this design. If the design does not cover the plausible input space, predictions made by the emulator at certain points in the space cannot be made to any sensible degree of accuracy. It is also important to consider the size of the training design, as it must be large enough to minimise the growth in uncertainty between points, as demonstrated by Figure 2.3, but must take into account the cost of evaluating the model multiple times.

If the approximation used in the emulator is simpler and more computationally efficient than the simulator, it offers a solution to the impracticality of applying UA and SA to complex models. The statistical functions employed for Gaussian process emulation are extremely fast to evaluate, and therefore match this criterion. A Gaussian process learns about the model as a whole, using all information from the local neighbourhood of a design point (Oakley and O'Hagan, 2002). As a result, measures produced by SA with emulators have less uncertainty than standard methods using the same design (O'Hagan, 2006). A considerably smaller set of points, and consequently fewer model runs are therefore required to build an emulator. Oakley and O'Hagan (2004) demonstrate the efficiency of emulator methods, performing SA using an emulator based on 250 runs and achieving the same level of accuracy as traditional methods performed using 15,360 simulator runs.

Approximating a simulator with a simple function creates potential for portability. Models generally require installation and configuration, a specific platform, and may be large binaries. An emulator is simply a set of training data and parameters, and assuming that standard mechanisms for representing and evaluating an emulator are available, they can be easily transported and evaluated with a variety of statistical or modelling platforms. This is especially important when evaluating a remote model over a network connection, as it is impractical to transfer large volumes of data. Instead, it may be more practical to simply transfer the emulator, and evaluate with data stored locally.

The suitability of emulator methods varies depending on the characteristics of the model we wish to emulate. For best results, the model must be deterministic — meaning that the same value of y is always produced for the same value of x . Emulation of stochastic models, where random y values are produced by multiple simulator runs with a fixed value of x , is possible (Boukouvalas et al., 2009), but is still a relatively new field.

In order to estimate the output at nearby values to a point x in the training set, emulators based on Gaussian processes require the simulator function $f(\cdot)$ to be smooth (Oakley and O'Hagan,

2002). If the function is rough, such as those seen in stochastic models, a Gaussian process cannot accurately predict the simulator behaviour between two points in the training design. Some numerical methods can be combined with the Gaussian process to emulate the variation or noise between points in the training design (Boukouvalas et al., 2009), such as those which use a smoothness parameter. The use of Gaussian processes also restricts the output to continuous values, with discrete valued variables requiring further development of the theory (Tulloch, 2013).

Model inputs are subject to uncertainty from measurement and lack of knowledge. Unless the given input x is part of the training set, emulation introduces an additional source of uncertainty as it must predict the corresponding value of $f(x)$ using interpolation. Whilst the emulator can be used as a direct surrogate for the simulator in UA and SA, treating the emulator mean as the output, a full analysis must account for sources of uncertainty, including those introduced by the emulator.

Emulator methods typically assume the model has a single output. If a single output is not provided by the model, then it is possible to instead emulate a transformed combination of the outputs (Rougier, 2008), for example aggregating monthly rainfall estimates to a single yearly average. As it can be difficult to recover the underlying outputs in the transformation, a simpler approach may be to build an emulator for each output. If the emulator for each output has good validation results, this approach generally yields acceptable results. However, as the error on each output is treated independently by the emulator, any correlation between output errors is lost. If the simulator produces highly correlated output errors, the multi-emulator approach will not be suitable. Creating multiple emulators can also be time consuming, involving setting parameters, training the emulator, and validation for each output. Building true multi-output emulators is possible (Conti and O'Hagan, 2010), but complex, and may require a large number of training runs.

Emulation techniques have been applied in several real use cases, successfully demonstrating increases in efficiency and accuracy when only using a fraction of the data (Kennedy et al., 2006). The process of building an emulator is complex, and currently only accessible to specialist statisticians. Existing literature focuses on the underlying processes and mathematics, with a distinct lack of discussion on how accessibility for emulators can be provided, such that they can be built by non-specialist users, be integrated with model workflows, and deployed on the Web. It has been proposed by O'Hagan (2006) that emulation technology should be accessible to a wide

variety of users through software. GEM-SA¹, a Windows tool for building an emulator, making predictions, and performing UA and SA, aims to provide this accessibility. The tool hides most of the complexities involved in emulation, but suffers from usability issues in handling data. For example, once a design is generated using the tool, it must be exported and converted into a format understood by the model. The same issue is encountered when importing the model outputs back into the tool to perform emulator training.

2.3 The Model Web

Traditional modelling and data analysis techniques involve installing software, collecting data from a variety of sources, preprocessing the data, and evaluating models on a local machine. With everything owned and operated locally, the process becomes very expensive and time consuming. Interoperability — the ability of diverse systems to exchange and use information — is one of several factors limiting the application of models. Without interoperability, communication between multiple models and data stores is challenging and often extremely limited. Terminology to describe common properties may vary between other modellers, researchers, and policy makers. These users also face restrictions on model access and availability. Increasing the availability of, and interoperability between, models and resources can help to improve modelling and data analysis in various domains (Zhao et al., 2012).

The Model Web is a generic concept envisioned to enhance model interoperability, ultimately providing better access to and interaction between models (Nativi et al., 2012). Consisting of a system of interoperable components, the Model Web utilises Web services to provide machines and users with access to models and data stores (Geller and Turner, 2007), and places no restrictions on the size and quantity of these components. Detailed modelling scenarios often depend on contributions from multiple disciplines. For example, predicting the effects of climate change on wheat yields may involve models from many domains, including climate, edaphology and economics. As a result, since its original definition for the purpose of ecological forecasting, the Model Web concept has been proposed across various domains.

The idea of interoperable models and data stores is nothing new, and there are many frameworks available for model integration (Bastin et al., 2013). However, interoperability has in the past only been achieved within small groups of services, often from the same organisation. The Model Web seeks to remove these boundaries, and has the advantage over other frameworks of

¹<http://ctcd.group.shef.ac.uk/gem.html>

utilising the existing Web infrastructure, allowing models and data to be shared across the globe, and integrating components from a variety of sources.

SOA is a methodology for providing software applications as interoperable services (Papazoglou, 2003). A SOA consists of reusable software components, each representing a modular block of well-defined functionality. By combining several components in a SOA, complex functions and workflows can be constructed. This can serve as the basis for distributed computing and an interoperable framework of collaborating applications. The Model Web can be considered as a SOA, with each model, or simulator, built as a software component. It may be desirable to decompose a complex model into several smaller pieces of functionality, providing that the smaller pieces form useful and reusable components. A data store is also built as a software component, and communicates with the model by passing data in a standard format.

While Web services are not a requirement for SOA, they are commonly how such architectures are implemented, and therefore fundamental to building the Model Web. Exposing a model on the Web presents many opportunities for sharing, reuse, and workflow composition. Implementation is platform and language agnostic, and updating a model only requires the installation on the service to be changed, as clients are only communicating with this single instance. The interoperability provided by the Web can allow us to build generic tools. If model's inputs and outputs conform to a set of data types, it may be possible to use any software that supports those data types, and neither the developer of the model or software need write specific code to interact with an individual model. A layer of abstraction is provided by service interfaces, allowing clients to specify input data and execute service operations without specialised knowledge of the underlying implementation. One of the major challenges faced in the construction of the Model Web is reaching a consensus on Web service communication protocols and data encoding. Available protocols and data encoding standards are discussed in Section 2.4.

Although still a relatively new concept, Model Web components have been developed by various parties. eHabitat, a model for assessing the relative importance of protected areas in Africa (Dubois et al., 2011), is one example of a Model Web component. The implementation of eHabitat as a Model Web component is designed to extend participation of the scientific community, and create usage opportunities in scenarios not originally considered when developing the model. For example, if the output from a climate change model is used as input data, eHabitat essentially becomes a service for assessing the impact of climate change in protected areas.

The eHabitat example stresses the requirement for uncertainty propagation in the Model Web.

When executing complex workflows involving several components, all of which are sources of uncertainty, the final output must provide information on the quality of the result, a matter which is increasingly important where intermediate results may be inaccessible (Skøien et al., 2011). Although uncertainty propagation support for the eHabitat component was implemented by Skøien et al. (2011), there are few examples demonstrating propagation in a workflow — from uncertain inputs, through several components, to representation in an uncertain output.

Full utilisation of the Model Web — widespread integration of independent, multidisciplinary models — is yet to develop. As discussed in Section 2.2, tools for performing SA on a model or building an emulators are currently restricted by software package or platform, and can require much pre-processing to transfer data between model and tool. The interoperability provided by the Model Web presents an opportunity to create Web-based tools which, as a result of standard interfaces and data formats, are compatible with an unlimited amount of models. The development and use of such tools is an unexplored area.

2.4 Technology for Web services

Section 2.2 introduced the idea of communicating with models and data stores via Web services. Formally, a Web service is a software component which supports communication with other machines over the Web, and is a typical implementation strategy for a SOA. Alternative technologies, Remote Method Invocation (RMI) and Common Object Request Broker Architecture (CORBA), can also be used to implement distributed architectures (Papajorgji et al., 2004). However, neither of these technologies have seen mainstream adoption, with RMI limited to the Java platform (Juric et al., 2006), and CORBA suffering from problems related to complexity and protocol readability.

The success of the Model Web depends on the level of interoperability that can be achieved. Due to the variability in systems, language and platforms, application of interoperability in computer systems poses a great challenge. The challenge will be met if models are able to communicate with each other in a standard way, requiring a set of suitable information models for representing data, and service interfaces for defining interactions. When interoperability is achieved, providing opportunities for data sharing, code reuse, and generic tools in the Model Web is possible.

Established standards ensure uniform methods, data structures, and behaviour, and their use on the Web helps to achieve interoperability between services, as conforming to a standard, or specification, guarantees that each service can integrate with other Web services and clients. The

Method	Description
GET	Retrieve information identified by the URI.
POST	Create a new entity linked to the resource identified by the URI.
PUT	Store an entity at the given URI.
DELETE	Delete the resource identified by the URI.

Table 2.1: A list of frequently used HTTP methods.

most commonly implemented Web services standards are those which are domain independent, and developed by the World Wide Web Consortium (W3C), but the Open Geospatial Consortium (OGC) also maintain a number of standards. OGC standards are for Web services of a geospatial nature, building on those developed by the W3C to add features specific to geospatial problems. Although not strictly a standard, services using a REST architectural style have recently been growing in popularity, opting to utilise the array of the features provided by the HTTP.

2.4.1 Domain independent

The World Wide Web Consortium (W3C) is an international community with members from commercial, educational, and governmental organisations. The W3C works to develop domain independent Web standards, including HTTP, XML, SOAP and WSDL, four complementary standards considered to be essential technologies for deploying and describing a Web service (Louridas, 2008).

HTTP

Hypertext Transfer Protocol (HTTP) is a request-response application protocol used for data communication in the World Wide Web, and the most widely-adopted protocol for Web services. The basic principle of HTTP is to apply a method to a resource identified by a Uniform Resource Identifier (URI). A summary of frequently used methods is shown in Table 2.1. A HTTP message contains a header and optional body. Headers are used to transfer parameters of the HTTP transaction, for content negotiation, caching and authentication schemes. The protocol can be extended with the addition of new methods and headers, such as COPY and MOVE for Web Distributed Authoring and Versioning (WebDAV). HTTP is extremely well-established, and detailed description of the protocol is outside the scope of this work. Further information can be found in Fielding et al. (1999).

XML

Originally standardised in 1998, Extensible Markup Language (XML) is a format for representing structured information (Bray et al., 2008). XML is simple, text-based, readable by humans and machines, and transferable both locally and across networks. XML is designed to support a wide variety of use cases, and provides the basis for a number of application and domain specific formats and protocols. Examples include Really Simple Syndication (RSS) for publishing frequently updated works, the SOAP protocol for exchanging messages between systems, and Geography Markup Language (GML) for representing spatially referenced geometries. An XML document consists of a number of elements, which can either contain other elements or primitive values. Attributes can be assigned to an element to provide information that is not part of the data, such as a part number of an engine (Listing 2.1).

```
<Car>
  <Model>Ka</Model>
  <Manufacturer>Ford</Manufacturer>
  <Engine partNo="F0424E">
    <Type>Petrol</Type>
    <Size>1248</Size>
  </Engine>
</Car>
```

Listing 2.1: An XML document representing a Ford Ka.

The usability of XML is enhanced by the ability to validate an instance document, allowing us to ensure an instance document conforms to a given specification. XML schema defines a set of constraints for an XML document, and is commonly used to define XML-based formats (Section 2.4.3). Documents are constrained by assigning each named element a type, which specifies what elements, attributes, values are allowed as content in that element. A type is regarded as complex if it contains other elements, or simple if it only contains primitive values, and can either be nested within an element definition, or named so it can be reused elsewhere. Figure 2.2 demonstrates the use of a nested type to specify a car, and a named type to specify a part, which is subsequently used to specify an engine.

To maximise flexibility and reuse, abstract definitions can be created with the XML schema type system. In Figure 2.2, a part is defined by an abstract type only specifying a part number, and the engine is a concrete extension of a part. Whilst abstract definitions can help to create formats for use across several applications and domains, they should be used with care. If a definition is too abstract, the range of possible legal elements may be too broad, and it may be impossible for

```
<xs:schema>
  <xs:element name="Car">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Part" type="partType" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- All parts have a code -->
  <xs:complexType name="PartType">
    <xs:attribute name="partNo" type="xs:string" use="required"/>
  </xs:complexType>

  <!-- Engine is a part with type and size -->
  <xs:element name="Engine">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="PartType">
          <xs:sequence>
            <xs:element name="Type" type="xs:string"/>
            <xs:element name="Size" type="xs:integer"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 2.2: An XML schema defining a car, used to validate Listing 2.1.

users to implement in their client or server software. In contrast, a concrete definition could lack the flexibility required for widespread usage. A careful balance between the abstract and concrete definitions is therefore necessary to ensure the correct level of flexibility and usability.

As element and attribute names in XML are defined by the developer, mixing XML documents from different applications, formats, and protocols may result in naming conflicts. To solve this, XML element and attribute names can be prefixed, and a namespace for the prefix must be defined. A namespace is specified by a URI, for the purpose of giving the namespace a unique name. For increased usability, the URI should ideally be a Uniform Resource Locator (URL) that resolves to a Web page containing information about the namespace. In Listing 2.3, namespace prefixes are defined in the root element to prevent naming conflicts between two Keyboard elements — one of the musical variety, and the other for computing.

```
<Stock xmlns:m="http://www.uncertamart.com/music" xmlns:c="http://  
www.uncertamart.com/computing">  
  <m:Keyboard>  
    <m:Keys>61</m:Keys>  
    <m:Voices>375</m:Voices>  
  </m:Keyboard>  
  
  <c:Keyboard>  
    <c:Layout>QWERTY</c:Layout>  
    <c:Interface>USB</c:Interface>  
  </c:Keyboard>  
</Stock>
```

Listing 2.3: The use of namespaces to prevent naming conflicts.

SOAP

Simple Object Access Protocol (SOAP) is a protocol for exchanging messages between systems, commonly used in network services (Box et al., 2000). Based on XML, it has been designed to be independent of programming language or other implementation detail. A SOAP message consists of a mandatory SOAP envelope, an optional SOAP header, and a mandatory SOAP body. The SOAP header is an extensible element used for transferring infrastructure information. Extensions such as authentication, transaction management and payment are typically implemented as header entries. The SOAP body contains either the actual message intended for the recipient, or a SOAP fault. The SOAP fault is a standard mechanism to transfer error information within a message. This information can help to identify the cause of the fault — for example, if the client provided incorrect details for authentication.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Could not find greeting module.</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Listing 2.4: A SOAP envelope containing a fault.

WSDL

Web Service Description Language (WSDL) is an XML specification for describing network services (Christensen et al., 2001). A WSDL document for a service defines abstract messages, operations and endpoints. A binding mechanism is then applied, attaching a specific protocol, data format or structure to these abstract definitions. The WSDL specification introduces binding extensions for SOAP, HTTP and Multipurpose Internet Mail Extensions (MIME), of which SOAP is the most common. A WSDL document consists of the following elements:

types contains XML schema data type definitions;

message a specification of a message sent to, or received from, a service;

portType a definition of a service, containing set of operations and their associated input and output messages;

binding binds operations to a specific protocol and data format;

port defines a single communication endpoint by specifying an address for a binding;

service groups a set of related ports.

These four standards aim to provide interoperability between computer systems. When combined, platform and language-neutral services can be published on the Web. The availability of tools and programming language support for XML is widespread, with software for creating and editing instance and schema documents, and a plethora of libraries for most languages. The self-describing nature of WSDL creates the opportunity for users to simply browse for a service they wish to use (Lim and Wen, 2003). Developers can use the concrete message descriptions provided in WSDL to automatically generate client code. This reduces implementation time and effort as a developer is not required to completely understand the technical details of a service to consume

it (Staab et al., 2003). These tools are available for a variety of languages, giving strength to the language and platform independent nature of communication with Web services.

Crasso et al. (2010) identify several factors potentially limiting third-party WSDL service consumption. Functionality is often hard to understand due to a lack of documentation provided in WSDL documents by service developers. WSDL only provides a technical description of a service, and not in a human-readable form. Although a service may be accompanied by external documentation, there is no standard way to reference this in a WSDL document. An issue with individual implementations, rather than the specifications themselves, is that SOAP and WSDL services may often use the same message structure for success and error responses. This is considered bad practice. Instead of using flexible message structures for representing both success and error responses, the WSDL document should associate a message to an operation fault. Using the same message structure may bypass error handling provided by client software and code generation tools.

WSDL may lack the necessary service description quality for discovery (OGC 05-007r7, 2007; Crasso et al., 2010). Newmarch (2004) also raises this issue, stating that WSDL only specifies a service on a syntactic level, rather than semantic. This is crucial for service discovery, as the requirements and capabilities of Web services cannot be fully determined from the syntactic level descriptions provided by a WSDL document. Akkiraju et al. (2005) address this issue by introducing WSDL-S, a method for annotating WSDL documents with ontologies.

Complexity and interoperability issues can occur due to the use of XML schema for WSDL message descriptions (Pautasso et al., 2008). XML schema is a very rich language, which can make it difficult to express a data model in a way that maintains support with multiple service implementations. The problem can be avoided by creating schema profiles — a subset of types that can be used in different scenarios. However, creating profiles requires a careful balance of flexibility and interoperability; if one restricts the usable types too severely the number of services able to use a profile will be limited.

REST

Representational State Transfer (REST) is an architectural style first introduced in Fielding (2000). REST architectures are based on the transfer and representations of resources, a contrast from the SOAP focus on Remote Procedure Call (RPC) and message transfer. Resources, and collections of resources are exposed on a server and identified with a URI. A client can trigger state transitions

of a resource by issuing requests to the server. REST is commonly described in the context of HTTP, taking advantage of the full range of methods.

Due to the use of existing standards, a perceived advantage of REST services is simplicity (Pautasso et al., 2008). In a REST architecture, all of the HTTP methods are used. POST, PUT, GET, and DELETE methods are typically employed to execute create, read, update, and delete operations on resources. In contrast, SOAP over HTTP typically only uses the POST method, requiring further operation definition in the request message. Clients, proxies and any other existing network components can support caching and security without requiring additional extensions. A vast number of HTTP libraries and tools are available for client and server development, and testing can be performed in an ordinary Web browser. While SOAP is limited to XML, resource representations in a REST architecture are unrestricted in type, making it possible to use lightweight data-interchange formats such as JSON.

The lack of interface specification for services adopting a REST architecture can be a disadvantage (Pautasso, 2009). WSDL and Web Application Description Language (WADL) have the capability to describe interfaces to REST services, but are rarely used. The simplicity provided by REST can alleviate some of these problems for client developers, as documentation can often be provided in a human-readable format, a benefit also shared by XML-based services. However, this is still a disadvantage in situations where a description is required in a machine-readable format, such as client generation tools and workflow orchestration software. Although the HTTP specification states that there is no limit on the length of a URI, this may not be true in reality. Pautasso et al. (2008) note that it is not possible to encode a large amount of data in a resource URI. The request should instead be sent using POST, which does not have these limitations. However, they do not consider that using POST may not be appropriate for the operation you wish to perform, hence abusing REST principles. For example, when using a GET request to retrieve a geospatial resource, a user may wish to restrict the results to a specific bounding box. The size of the bounding box data may be too large for the GET query string, thus forcing the use of POST.

2.4.2 Geospatial specific

The Open Geospatial Consortium (OGC) define a number of standards for geospatial Web services. The application specific standards defined by the OGC utilise more generally applicable technologies created by the W3C. Work has shown that these standards provide a platform for service interoperability within a spatial data infrastructure (Christensen et al., 2006), leading to

their adoption by organisations in several countries (Zhang et al., 2008). Standards developed by the OGC include the following Web service interfaces:

Catalog Service for the Web (CSW) for discovering, browsing, and querying metadata about data and services;

Sensor Observation Service (SOS) for querying observations, sensor metadata, and representations of observed features;

Web Coverage Service (WCS) for retrieval of coverages — geospatial information representing phenomena that varies over space/time;

Web Feature Service (WFS) for access and manipulation of geographical features;

Web Processing Service (WPS) for the execution of geospatial processes.

The WPS standard aims to facilitate the publishing, discovery, and client binding of geospatial processes on the Web. A process is defined as any algorithm, calculation, or model that performs an operation on spatially referenced data (OGC 05-007r7, 2007). WPS is the standard of most interest for building the Model Web, as in this case the geospatial process will be a model. The standard defines an XML-based client-server communication protocol with three main operations:

GetCapabilities to return metadata about the service, including the processes offered;

DescribeProcess to retrieve information about a specific process, including its inputs and outputs;

Execute which includes any required inputs and parameters, and returns the output(s) of the process.

The common usage pattern for a WPS consists of firstly discovering a service endpoint — either through a catalogue or other means. A list of available processes can be retrieved by sending a *GetCapabilities* request to the discovered endpoint. For an individual process, the *DescribeProcess* operation returns a specific list of inputs/outputs and their associated data types. With this information the client can gather the required data, and build and issue an *Execute* request. Once the request is complete, the client will receive either the result from the service, or a standard exception message if the process could not be executed.

The standardised WPS interface is designed to be generic, meaning processes offered by a WPS can range from a simple addition of spatially referenced numbers, to simulations on a global

scale. The specification includes valuable features to help describe geospatial processes and their inputs/outputs, execute and control a process, and handle output from a process. If the standard is implemented correctly, the service consumer has a guarantee that they are able to include these features in a request.

Many geospatial processes will be slow to execute, and model simulations may take hours, if not days to run. Traditional requests over HTTP keep an open connection until a response is received. A request for execution of a long running process may result in a time out, and there is always the risk of an interruption to the connection, meaning the response will never be received. To prevent this, the WPS defines a mechanism for asynchronous execution. Upon receiving the process execution request, the WPS instance will immediately return a response containing a URL. This URL locates the process result, or if the process is still active, execution progress information.

A WPS process developer is required to specify a minimal amount of metadata for inputs, outputs, and the process itself. To further improve process discovery and use, they can also specify additional properties such as the units of measure, supported Coordinate Reference Systems (CRSs), and a set of allowed values.

```
<Input minOccurs="0" maxOccurs="1">
  <ows:Identifier>Domain</ows:Identifier>
  <ows:Title>The prediction domain</ows:Title>
  <ows:Abstract>
    The prediction domain specifies the locations at where you wish the algorithm to
    predict, it can be a GML Point, Polygon, MultiPoint, MultiSurface or RectifiedGrid
  </ows:Abstract>
  <ComplexData>
    <Default>
      <Format>
        <MimeType>text/XML</MimeType>
        <Schema>http://schemas.opengis.net/gml/3.1.1/base/gml.xsd</Schema>
      </Format>
    </Default>
    <Supported>
      <Format>
        <MimeType>text/XML</MimeType>
        <Schema>http://schemas.opengis.net/gml/3.1.1/base/gml.xsd</Schema>
      </Format>
    </Supported>
  </ComplexData>
</Input>
```

Listing 2.5: Extract from a WPS *DescribeProcess* response, showing the description of an input.

SOA concepts are supported by the ability to provide any input or output data as a reference. Instead of an embedded payload in the request or response XML documents, a URL where the

data can be located is given. This enables retrieval from, or publishing to, data interfaces such as WFS and SOS. Embedding binary datasets encoded as American Standard Code for Information Interchange (ASCII) strings in an XML document greatly increases the size of the data, making referenced data support crucial.

WPS software development within the open source community has been active. Service frameworks such as 52°North WPS², PyWPS³, and the ZOO Project⁴ have been developed to implement the standard, and GRASS GIS⁵ has support for generating WPS compliant process descriptions. However, client support is still limited. For example, a uDig⁶ plugin is able to send requests to a WPS, but only a very limited number of input and output data formats are supported. If a format is unsupported, the user is unable to use the process. A uDig plugin for orchestrating workflows has also been developed (Schäffer and Foerster, 2008), but is only compatible with WPS instances and shares data format issues with the client plugin.

The limited client support may be a factor in the poor adoption rates detailed by Lopez-Pellicer et al. (2012). Although version 1.0 of the WPS standard was published in 2007, out of a total of 9329 OGC Web services discovered in March 2011, only 58 were WPS instances. As expected, a large number of these services were hosted by WPS framework and specification developers. Many of the service instances discovered did not support WSDL. This is unsurprising as the WPS specification is vague on how a WSDL document for a WPS instance can be created. There was also a lack of metadata and descriptions offered by the services, potentially limiting the prospects of user discovery. However, this is a problem not only limited to processes exposed using the WPS standard.

In addition to the lack of client support, there are a number of possible explanations for the low number of services. The study by Lopez-Pellicer et al. (2012) did not include services registered in catalogues. However, as the CSW specification does not yet support WPS (OGC 07-006r1, 2007), this would have likely had a minimal impact on the result. Implementing geospatial processes on the Web can be complex, and whether people consider the effort worthwhile is a key factor affecting Web service adoption in general.

WPS has been found to be workable in the case of replicating traditional client-side Geographic Information System (GIS) functionality (Michaelis and Ames, 2009), albeit with some

²<http://52north.org/communities/geoprocessing/>

³<http://pywps.wald.intevation.org/>

⁴<http://www.zoo-project.org/>

⁵<http://grass.fbk.eu/>

⁶<http://udig.refractory.net/>

shortcomings, such as the inability to cancel a submitted job. However, the wider applicability and usability of the WPS is not considered, and alternative solutions are rarely mentioned. The WPS is considered to be the specification of most interest in developing the Model Web (Nativi et al., 2012), but the suitability of the specification for supporting such a diverse range of models and interoperability challenges remains untested.

The OGC Web service standards primarily communicate with XML, and support for SOAP has been added to newer specifications. However, the OGC do not use the W3C developed standard, WSDL, for service descriptions. WSDL focuses on explicit interface description, listing operations, inputs, outputs and payload types (OGC 08-009r1, 2008). The OGC standard method, *GetCapabilities*, lists operations but not messages and payload types. OGC Web service descriptions contain additional metadata and geospatial specific features such as bounding boxes and available layers, which have the potential to aid service discovery. The previously mentioned WSDL-S extension could help to bridge the gap between the syntactic level of detail in a WSDL file, and the semantic level provided by in OGC service descriptions.

Consuming services is straightforward with SOAP/WSDL. WSDL documents are of a technical nature and contain concrete schema message descriptions, creating the possibilities for automatic code generation and generic execution clients. Automatic code generation can greatly accelerate development of integration with services in both existing and new software. In comparison to this approach, the WPS is complicated for the consumer. As message descriptions are not provided in a standard format such as XML schema, the consumer must have knowledge of the WPS specification. As previously detailed, the pattern for WPS usage is complex, and involves three operation calls: *GetCapabilities* to list process identifiers, *DescribeProcess* to retrieve a description of the desired process, and *Execute* to start the processing and receive the result.

During the initial development of OGC Web service specifications, WSDL was not a formally approved standard (Padberg and Greve, 2009). Version 1.0 of the WPS specification was finalised in 2007, several years after the SOAP and WSDL standards originally defined. With these existing, popular technologies well established, it is unclear why the OGC failed to integrate them into their own standards. An attempt was made for WPS, but integration detail in the specification is lacking clarity. No concrete examples are provided, and practices suggested for the WSDL request message structures do not conform with structures defined elsewhere in the specification.

The ambiguities in the specification were partially resolved in OGC 08-009r1 (2008), detailing two approaches for creating a WSDL document for a WPS instance. The first approach describes

all process with a single WSDL document, whilst the second approach describes each process with an individual document. The use of a single WSDL document simply describes the three WPS operations: *GetCapabilities*, *DescribeProcess*, and *Execute*. As no description is given on the processes available on that instance, this approach is only viable in a limited number of scenarios. The user is required to have prior knowledge of the processes offered by the instance, and how to construct a process execution request document. Creating an individual WSDL document for each process appears have greater applicability — the descriptions contain the complete list of inputs and outputs. However, the approach is still ambiguous. Execution requests that are generated in accordance with the message descriptions in the WSDL adopt process-specific element naming, and therefore do not comply with those defined in the WPS specification. As a result, standard features of the WPS specification, such as reference passing and asynchronous invocation, are unavailable unless support for such non-standard messages is implemented on the server. Until these matters are resolved, WSDL support for WPS remains an issue for further research.

Many attempts have been made to add WSDL compatibility to WPS. An approach by Brauner and Schäffer (2008) derives WSDL documents dynamically from WPS instance data by transforming the minimal WSDL descriptions provided by the specification. In the absence of SOAP from some WPS implementations, basic HTTP POST bindings are used to create a valid WSDL document. However, the automatically generated WSDL documents have limited usability. They do not describe the inputs and outputs for a process, requiring the user to know what processing capabilities the instance has. The responsibility of implementing this approach rests on the owner of the WPS instance. Unless part of an existing WPS framework, service owners may not be willing to invest time in supporting automatically generated WSDL documents. Especially considering it is not a requirement of the WPS specification.



Figure 2.4: An overview of how the WPS proxy service proposed by Sancho-Jiménez et al. (2008) operates.

A proxy service proposed by Sancho-Jiménez et al. (2008) moves the responsibility away from the service owner. An overview of how the proxy service operates is shown in Figure 2.4, which consists of the service issuing *GetCapabilities* and *DescribeProcess* requests to build a WSDL document for a specific WPS instance. In contrast to the limited process descriptions generated in

Brauner and Schäffer (2008), proxy WSDL documents describe each input and output. A message specified in a WSDL document must refer to an element, or set of elements, within a schema. However, WPS describes an input or output with only a MIME reference and reference to an XML schema. As a result, the proxy service cannot specify a concrete type, instead resorting to the *anyType* element.

```
<!-- Assume GML schema has been imported -->
<xsd:element name="Domain" minOccurs="0" maxOccurs="1">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="gml:Point"/>
      <xsd:element ref="gml:Polygon"/>
      <xsd:element ref="gml:MultiPoint"/>
      <xsd:element ref="gml:MultiSurface"/>
      <xsd:element ref="gml:RectifiedGrid"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Listing 2.6: Extract from a WSDL document, showing the XML schema for an input.

The geospatial community has made a move to adopt REST architectures for data services, such as that for the OGC Web Map Tile Service (WMTS) (OGC 07-057r7, 2010). A similar interface for the WCS has also been proposed (Mazzetti et al., 2009). For data services, there is an inherent ease of mapping objects to resource URIs, and operations on those objects to HTTP methods.

In contrast, existing examples of REST processing services are limited. Foerster et al. (2011) demonstrates how such a service can be exposed using REST, but does not identify benefits for a processing service over other architectures. In the approach proposed by Foerster et al. (2011), metadata and process execution are exposed as resources which can be accessed using HTTP methods. The approach is simple, but inconsistent with the OGC service model. The request/response nature of processes means that the full range of HTTP methods is only utilised when dealing with asynchronous processing jobs. A job is created with a POST request and queried with subsequent GET requests which ultimately return the result when it becomes available. The job and its results can then be removed from the service using a DELETE request. Although introduced by the paper as being key concepts for REST, it remains unclear how to specify inputs and how processes themselves should be represented as resources. The example poses the question of whether the REST architecture is appropriate for every use case.

2.4.3 Data encoding formats

SOAP, WSDL and WPS are specifications for Web service interfaces, ensuring services describe their operations, inputs, outputs in the same way, and communicate using standard machine-to-machine protocols. They make no assumptions about the types of data used for inputs and outputs, allowing for an infinite amount of possible data types. We can restrict these data types to a limited, specific set by adopting standards for data encoding. Many data encoding standards exist, with the OGC defining GML and O&M, for encoding geospatial primitives and observations respectively. Also of note is UncertML for encoding various representations of probabilistic uncertainty.

GML

Geography Markup Language (GML) is an XML encoding standard for the description and transport of geographical information (OGC 07-036, 2007). GML contains an extensive set of types, including those for representing geometries such as points (Listing 2.7), lines, and polygons, time, units of measure and coordinate reference systems. These elements are commonly used as primitives in an application schema — containing definitions of object types specific to the domain of interest. For example, an application schema for city planning may define objects for roads, rivers, railways, and buildings. The standard has been widely implemented, and is adopted by all OGC specifications.

```
<gml:Point gml:id="point" xmlns:gml="http://www.opengis.net/gml/3.2" srsName="http://www.opengis.net/def/crs/EPSG/0/27700">  
  <gml:pos>216521.28107019 771211.422979205</gml:pos>  
</gml:Point>
```

Listing 2.7: A GML document representing a point.

O&M

Observations & Measurements (O&M) is a conceptual model and corresponding XML implementation for exchanging information describing observations (OGC 07-022r1, 2007). An observation is defined as an event which produces a value for an observed phenomenon at a specific point in time — whether that be the current air temperature, or the size of a field. The conceptual model aims to represent not only the resulting value, but additional information about the observation, and contains the following elements:

procedure describes the process used to produce the result;

resultTime is the time the procedure was performed;

observedProperty describes the phenomenon for which the result is an estimated value of;

phenomenonTime is the time the phenomenon occurred;

featureOfInterest represents the real-world subject of the observation;

result is the value produced by the procedure.

O&M documents typically import elements from GML to represent time instants, time periods, and feature of interest geometries. Listing 2.8 gives an example O&M observation, where the feature of interest is located at the same point represented in Listing 2.7. A scalar result is shown in this example, but the result of an observation is unrestricted in type, making O&M appropriate in a wealth of use cases (OGC 07-041r1, 2007; Raape et al., 2010). Whilst enabling a wide array of usage opportunities, such genericity can create interoperability issues, as a client or service will not simply be able to advertise support for O&M, as in reality they can only support a very small subset of results.

UncertML

Uncertainty Markup Language (UncertML) is a conceptual model, dictionary and encoding specification that can be used to convey probabilistic representations of uncertainty in an interoperable manner. A range of uncertainty types are supported by UncertML, from simple summary statistics such as mean and variance, to multivariate distributions and random samples. When dealing with uncertain observations, the generic nature of O&M allows us to use UncertML as the result in favour of a single scalar value.

UncertML was originally developed as a weakly typed language, where technically any type of statistic, distribution, or realisation could be encoded. This approach involves the use of generically named elements with the exact types specified with an attribute, which links to a dictionary definition (OGC 08-122r2, 2007), and allows flexibility at the cost of usability. The weakly typed design allows a user to reference their own uncertainty types, and does not encourage the use of a shared dictionary. This poses a major problem for interoperability, as this could lead to the use of multiple definitions for well-known types, such as the normal distribution or variance. If the definitions of two types do not match, they will be treated as different representations of uncertainty.

These problems were addressed by the second version of UncertML, which shifted to a strongly typed language, where each supported uncertainty type has a named element (Bastin and Williams,

```

<om:OM_Measurement gml:id="measurement" xmlns:om="http://www.opengis.net/om/2.0"
xmlns:gml="http://www.opengis.net/gml/3.2" xmlns:xlink="http://www.w3.org/1999/
xlink" xmlns:sams="http://www.opengis.net/samplingSpatial/2.0" xmlns:sa="http://
www.opengis.net/sampling/2.0">
  <om:type xlink:href="http://www.opengis.net/def/observationType/OGC-OM/2.0/
OM_Measurement"/>
  <om:phenomenonTime>
    <gml:TimeInstant gml:id="time">
      <gml:timePosition>2012-01-01T00:00:00.000Z</gml:timePosition>
    </gml:TimeInstant>
  </om:phenomenonTime>
  <om:resultTime xlink:href="#time"/>
  <om:procedure xlink:href="field_sample"/>
  <om:observedProperty xlink:href="area"/>
  <om:featureOfInterest>
    <sams:SF_SpatialSamplingFeature gml:id="feature">
      <gml:identifier codeSpace="http://www.uncertweb.org">1</gml:identifier>
      <sa:type xlink:href="http://www.opengis.net/def/samplingFeatureType/OGC-OM/2.0
/SF_SamplingSurface"/>
      <sa:sampledFeature xsi:nil="true"/>
      <sams:shape>
        <gml:Point gml:id="point" srsName="http://www.opengis.net/def/crs/EPSSG
/0/27700">
          <gml:pos>216521.28107019 771211.422979205</gml:pos>
        </gml:Point>
      </sams:shape>
    </sams:SF_SpatialSamplingFeature>
  </om:featureOfInterest>
  <om:result uom="m2">219726.98216837645</om:result>
</om:OM_Measurement>

```

Listing 2.8: An O&M Measurement representing the area of a field.

```

<!-- UncertML version 1 -->
<un:Distribution xmlns:un="http://www.uncertml.org" definition="http://
dictionary.uncertml.org/distributions/gaussian">
  <un:parameters>
    <un:Parameter definition="http://dictionary.uncertml.org/distributions/
gaussian/mean">
      <un:value>28.62</un:value>
    </un:Parameter>
    <un:Parameter definition="http://dictionary.uncertml.org/distributions/
gaussian/variance">
      <un:value>2.49</un:value>
    </un:Parameter>
  </un:parameters>
</un:Distribution>

<!-- UncertML version 2 -->
<un:NormalDistribution xmlns:un="http://www.uncertml.org/2.0">
  <un:mean>28.62</un:mean>
  <un:variance>2.49</un:variance>
</un:NormalDistribution>

```

Listing 2.9: A normal distribution encoded in UncertML versions 1 and 2.

2010). While it is now restricted to a finite set of uncertainty representations defined by the authors, usability has increased substantially as a result, as seen by comparing two versions of the same normal distribution (Listing 2.9). The common dictionary has been made available through the UncertML Web site.

JSON

With universal support in browsers, JavaScript is the client-side language of the Web. The language is essential for building powerful Web applications — those which only require a browser to run. A growing trend towards Web applications has fostered JavaScript Object Notation (JSON) as an alternative data interchange format to XML. JSON is based on a subset of the JavaScript programming language, and can be natively parsed into JavaScript objects. In contrast, XML requires specific parsers for each XML-based language, requiring greater implementation cost for a Web application developer over JSON.

```
{ "type": "Point",  
  "coordinates": [216521.28107019, 771211.422979205],  
  "crs": {  
    "type": "name",  
    "properties": {  
      "name": "http://www.opengis.net/def/crs/EPSG/0/27700"  
    }  
  }  
}
```

Listing 2.10: A geospatial point represented as a GeoJSON object.

JSON is built on two structures: an object, a collection of key/value pairs, and an array, an ordered list of values. The values in an object or array can be other objects or arrays, strings, numbers, boolean and null values. As objects and arrays map logically to typical programming language structures, the benefits of JSON are not limited to the JavaScript language. Listing 2.10 demonstrates the use of objects, arrays, and values to represent a geospatial point.

Popular social networking sites Twitter and Foursquare both deprecated parts of their XML APIs in late 2000, signifying the preference for JSON over XML for such Web applications. The geospatial domain has also made a move to adopt JSON, with the creation of several application and domain specific formats. GeoJSON is an open format for geometry and feature description, and shares much in common with the basic GML elements. Support for GeoJSON is widespread, with Web-based mapping libraries such as OpenLayers and Leaflet both able to import geometries and features encoded as GeoJSON. Although primarily XML-based formats, JSON encodings for

O&M and UncertML have recently been developed (Gerharz et al., 2011; Cornford and Williams, 2011).

2.5 Web-based workflows

A workflow is the automation of a business process, where a business process is a set of structured activities working collaboratively to achieve a business objective. In a workflow, a set of procedural rules define how information is passed from one activity to another to achieve the overall objective (Hollingsworth, 1995). If we consider the execution of a model or data retrieval an activity, we can build business processes, and subsequently workflows, for complex scenarios where the objective may be to produce forecasts, simulations, or provide important information for a decision making process. A service chain is defined as “a sequence of services where, for each adjacent pair of services, occurrence of the first action is necessary for the occurrence of the second action” (Percivall, 2002). Due to their similarity, we will consider ‘chain’ to be synonymous with ‘workflow’.

Through building complex workflows involving models from multiple disciplines, we can answer more questions than a single model (Dubois et al., 2011). The Model Web aims provide increased accessibility and sharing through interoperability, facilitating the creation of model workflows.

The lifecycle of a workflow consists of four stages: discovering relevant data and model services; composing a workflow with those services; executing the workflow; and analysing the results (Tan et al., 2009). If the location of data stores and models is unknown, service discovery must be performed. Discovery is typically performed by querying a catalogue, such as CSW for geospatial services (OGC 07-006r1, 2007), specifying parameters such as keywords, time periods, and spatial extent. Universal Description, Discovery and Integration (UDDI) is a similar mechanism for WSDL-based services, but has limited scalability and efficiency (Benson et al., 2006). Web service discovery is a major research topic, and is outside the scope of this work.

Composition begins by importing the services you wish to be contained in the workflow. The flow of the workflow can then be specified, adding any necessary execution conditions — for example only executing certain services if the output of another matches a given value. Creating links from the output of one service to the input of another may require some form of intermediate translation if the data types do not match, adding further complexity to the workflow composition process. The process can be assisted by with the use of a GUI (Figure 2.5), automated computer

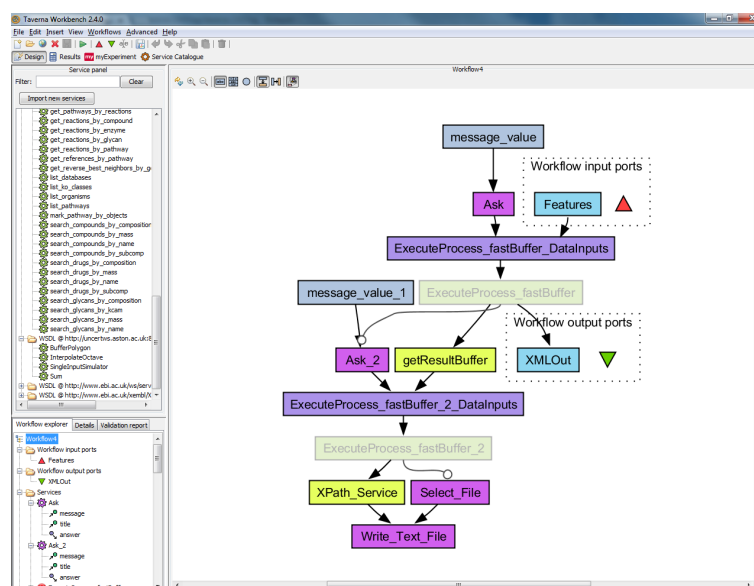


Figure 2.5: Composing a workflow in Taverna.

code, or performed by editing a human and machine readable document describing a workflow.

The workflow then be executed, commonly by an engine. A workflow execution engine is responsible for following the rules defined by the composition stage, and executing each action or service in turn. In the absence of an engine, the composition and execution steps are combined, and the services are executed as they are added to the workflow. Analysis involves both the intermediate service results, and the final composite output. This is typically an iterative process, where input parameters may be tweaked before running the workflow again, or the user may revert to the composition stage and include or exclude services.

The creation of complex workflows is only possible if model and data services are interoperable, and tools and software are available to orchestrate these workflows. Section 2.4 discussed technology to provide interoperability between Web services, and the remainder of this section focusses on workflow standards and software (Section 2.5.1), and the integration of such technology in the geospatial domain (Section 2.5.2).

2.5.1 Standards and software

A workflow system has a basic requirement for two components: a language to define the rules for how information is passed between services, and a software component to orchestrate the execution of these rules.

Element	Purpose
partnerLinks	Contains a list of participating services.
variables	Contains a list of messages and XML documents.
receive	Receives input from the workflow executor.
sequence	Defines a set of activities defining the flow of messages between services.
assign	Assigns data to and from variables used for service invocation.
invoke	Invokes a service with the message assigned to a given input variable.

Table 2.2: A list of commonly used BPEL elements.

BPEL

Business Process Execution Language (BPEL), a language defined by the Organization for the Advancement of Structured Information Standards (OASIS) for specifying peer-to-peer interaction between Web services, satisfies the former of these requirements. Generally regarded as an industry standard for Web services composition (Meng et al., 2009), and although originally designed for business workflows, BPEL can be used in any domain where Web service orchestration is required (Louridas, 2008).

BPEL is XML-based, and defines elements for the specification of workflow parameters, the sequence of events, and data flows between services (Table 2.2) (OASIS, 2007). Each BPEL document represents a single workflow, and a typical service execution flow consists of: assigning a request message to a variable; copying data from other variables to the request; and invoking the service. When invoking the service, the response is assigned to an output variable, which can be copied to the request message for a subsequent service execution. A rich array of control and flow elements can be included in a BPEL document, supporting conditions, iteration, and parallel execution. BPEL requires Web services to be described by a WSDL document, potentially limiting the usability of the language.

Due to the number of available elements and XML verbosity, creating a BPEL document can be complex and time consuming. Oracle BPEL Process Manager and Eclipse BPEL Designer are two graphical tools to facilitate the creation of workflows. These tools may still however be too complex for someone unfamiliar with Web service technology, as they require a degree of XML knowledge to assign input and output variables. The execution of a BPEL workflow is supported by several software components, including the Apache Orchestration Director Engine (ODE) and ActiveBPEL orchestration engines. Once a workflow is deployed on either of these engines, it is published as a SOAP/WSDL Web service, which can then be executed from a client, or even another workflow.

Taverna

A solution to the complexity of BPEL is offered by Taverna Workbench, an open source desktop client application for designing and executing workflows. Taverna allows the integration of several different components in a workflow, including SOAP/WSDL and REST Web services. The graphical interface provided allows users to add services to workflows, modify the order of execution, and link service inputs and outputs. Taverna attempts to hide much of the underlying complexity of Web service composition, instead focussing on usability for scientists, decision makers, and those unfamiliar with Web services and programming (Oinn et al., 2004).

By default, workflows are executed on a local machine. In scenarios where this is infeasible, such as when a workflow includes long-running processes, jobs can be remotely executed on a Taverna Server installation. Taverna is the implementation of a workflow language, composition GUI, and execution engine. Instead of adopting BPEL, Taverna represents workflows in a Simple Conceptual Unified Flow Language (SCUFL) document, a similar XML-based format.

A fundamental concept of the Model Web, and any SOA, is component reuse. While BPEL engines support this by exposing workflows as Web services, the local installation nature of Taverna does not support such functionality. Instead, the Taverna community facilitates workflow sharing and reuse through myExperiment⁷, an environment for finding, sharing, and rating workflows. Although this solution lacks in flexibility in comparison with the Web services exposure offered by BPEL, the central repository offered by myExperiment does create opportunity for workflow discovery.

In evaluating the suitability of BPEL and Taverna scientific workflows, Tan et al. (2009) conclude the latter to provide better support for the complete workflow lifecycle. This conclusion is somewhat unexpected, as BPEL is only a workflow specification language, whereas Taverna is fully-featured tool suite. Consequently, BPEL relies on community efforts to build tools to support workflow composition and execution, and currently these tools are not substantial enough for a non-technical user. BPEL does have some advantages over Taverna in the degree of control, and engines hosted on remote servers. BPEL allows control over parallel and conditional execution, and iteration, as opposed to the data-centric flow in Taverna where services are simply executed whenever their input data is available, and iteration is implicit depending on the input types. While Taverna workflows are mostly executed locally, BPEL execution engines are typically hosted on remote servers, providing increased reliability and scalability. When running long and computa-

⁷<http://www.myexperiment.org/>

tionally expensive jobs this is especially important, as execution failure will result in lost time.

2.5.2 Geospatial service integration

Standards and software for composing and executing workflows, such as BPEL and Taverna, are generally compatible with Web services built using domain independent technologies, including SOAP and WSDL. In the geospatial domain, the OGC decided against adopting SOAP and WSDL, and developed a separate approach for their Web service interfaces. These interfaces include the WPS, which is of greatest relevance in deploying models on the Web. This alternative approach therefore presents a challenge when integrating geospatial services into workflows.

The WPS specification contains several features to facilitate the inclusion of processes in a workflow (OGC 05-007r7, 2007). These features are utilised by Stollberg and Zipf (2007, 2008a,b), who propose a number of different mechanisms for composing and executing WPS workflows. The first of these approaches employs a single, composite process to act as a workflow orchestrator. This process is responsible for sequentially calling each service in the workflow, and handling input and output matching. In this approach, the workflow is exposed as a standard WPS process, supporting execution from existing clients and tools, and does not require the overhead of an orchestration engine. However, the static nature of the composite process involves coding the logic for service calls and data matching, compiling that code, and deploying on a WPS instance. Any modifications to the workflow involve repeating these tasks, greatly reducing the reusability and flexibility provided by workflows. In many cases workflow composers are unlikely to have the technical knowledge required to perform such tasks, therefore making this mechanism infeasible for all but a small minority.

As an alternative to HTTP POST requests with an XML message, the WPS specification provides an optional key-value pair (KVP) encoding for process execution requests. A KVP encoded request specifies parameters in the query string part of a URL, including the process identifier, inputs, and the desired format of the output. Figure 2.11 shows an example KVP execution request for the 'buffer' process, with inputs 'Geometry' and 'Distance', and a single output 'BufferedGeometry'. As it would be impossible to encode complex data sets in a query string, KVP instead relies on Web accessible references.

Stollberg and Zipf (2007) exploit KVP encoded requests to create a cascading service chaining mechanism. If we construct a KVP encoded request to one process, it is possible to use this request as an input data reference in another processing request, thus creating a chain. At runtime, the WPS

```
http://example.com/?
  service=WPS&
  version=1.0.0&
  request=Execute&
  Identifier=buffer&
  DataInputs=Geometry=@xlink:href=http%3A%2F%2Fmydata.com%2Froad.xml;Distance=20&
  RawDataOutput=BufferedGeometry
```

Listing 2.11: A KVP encoded process execution request.

instance issues an HTTP GET request to the data reference URL, which in this cases is a KVP encoded request. The calling WPS instance is unaware that the URL triggers a process execution request, and simply waits for a response.

The cascading chaining mechanism shares the same strengths as the composite process — the approach does not require an orchestration engine, and existing software can be used. The forced use of references reduces the data transferred between client and server, but does require all complex inputs to be exposed on the Web, something which may not always be possible. Drawbacks also exist in the composition of such workflows, as the length of the KVP request URL will be substantial for complex workflows involving several processes. Reusability is limited as the workflow only exists as KVP encoded WPS request, and must be modified manually for each subsequent request involving different input data.

For the final mechanism, Stollberg and Zipf (2008b) develop a simple XML language to define a WPS process workflow. The language provides several elements for specifying workflow components and characteristics, including the services, processes, execution order, and input/output matching. The language has many similarities to BPEL, which the authors consider too complex for non-technical GIS users. However, they fail to note that the complexities of BPEL can be alleviated with an appropriate client. A WPS process, acting as an orchestration engine, is deployed to execute workflows specified in this language. While the language is similar to, but less complex than BPEL, adoption will be limited by its proprietary nature and restrictions to WPS services. Exception handling, a feature all three approaches lack, is critical in Web service workflows. As a workflow can involve executing a multitude of services, execution failures must be handled and reported to the user, with an indication given as to the responsible service, and the cause.

With the quantity of tool and engine support for BPEL, alternative mechanisms may seem time consuming and unnecessary — consider that the effort to define a new orchestration language may have been better spent creating a client to hide the complexity of BPEL from a user. However, the motivation for these mechanisms has surfaced from disadvantages in orchestrating geospatial Web

service workflows with BPEL (Stollberg and Zipf, 2008b).

Although workflows involving WPS instances are possible (OGC 05-007r7, 2007; Meng et al., 2009), implementation of such workflows involves additional effort and several compromises. In BPEL, all included Web services must be described by with WSDL, meaning a WSDL document for each WPS instance must be defined. The lack of requirement for WSDL in the WPS will often mean this will not be done by the process developer (Lopez-Pellicer et al., 2012). As reviewed in Section 2.4.2, an automatic WSDL generation proxy (Sancho-Jiménez et al., 2008) could potentially minimise the effort required to provide WSDL documents for WPS instances. However, the input/output weak descriptions contained in the generated WSDL documents must be tested for their suitability in BPEL workflows.

In an existing example, Meng et al. (2009) describe geospatial service orchestration, but do not provide implementation detail — an important oversight considering the aforementioned complications surrounding WSDL and WPS integration.

The Apache ODE and ActiveBPEL orchestration engines expose a workflow on the Web as a SOAP/WSDL service. A more appropriate option in the geospatial community may be to instead expose the workflow on a WPS. Schäffer and Foerster (2008) propose a transactional extension to the WPS specification, enabling BPEL workflows to be dynamically deployed, and then available as a WPS process. However, considering the wealth of support for SOAP/WSDL services compared to those for WPS, the benefits of providing such an interface are unclear. In a similar manner, Yu et al. (2012) suggest extending a BPEL engine to support directly communicating with OGC Web services without the need for WSDL.

Composing Web service workflows with BPEL remains inaccessible for non-technical geospatial, especially considering the data formats in use (for example, GML and O&M), as input/output matching may require complex sub-selection. As previously identified, Taverna may be a better solution for non-technical users. Taverna was originally primarily aimed at the life sciences community, and therefore does not provide explicit support for OGC Web services. It does however support domain independent Web service technologies, such as SOAP and WSDL. Consequently, we are faced with the same challenges encountered when integrating WPS with BPEL.

Successful orchestration of WPS workflows with Taverna is demonstrated by de Jesus et al. (2011), who extend the basic SOAP and WSDL structures described in the WPS specification (OGC 05-007r7, 2007). WSDL documents are automatically generated from a PyWPS implementation, and contain enough information for Taverna to present the list of processes and associated in-

puts and outputs to the user. Additional support for asynchronous process execution through SOAP/WSDL, a matter undocumented in the WPS specification, is developed. Although the example proves the possibility and feasibility of WPS workflow orchestration with Taverna, the full potential of the software cannot be exploited. The generated WSDL documents do not provide substantially strong enough descriptions of inputs and outputs, a drawback faced by similar WSDL generation mechanisms. The reviewed approach is also specific to a single implementation, the Python-based PyWPS, and does not consider process developers not familiar with the language.

2.5.3 Uncertainty in workflows

When considering a workflow consisting of several autonomous processes, the issue of uncertainty becomes even more critical. If uncertainty is not propagated throughout the entire set of processes, it is impossible to say anything about the quality of the result. The responsibility of uncertainty management in a workflow can belong to a variety of parties. If uncertainty is supported in the input, a process is responsible for returning an uncertain response. When a process does not support uncertainty in the input, the responsibility could fall to either the workflow creator, or orchestrator. In both cases, it must be ensured that the uncertain output from one process is sampled as required, and the subsequent process executed for each sample.

2.6 Summary

This chapter reviewed the concepts of modelling and uncertainty management, the technologies for integrating models with the Web, and the construction of complex workflows involving these Web-based components.

The importance of uncertainty in the modelling process was explored, recognising sources of uncertainty in input observations, and also the underlying model structures. Providing complete information for the decision making process requires propagation of uncertainty, from input through to model output, typically involving sampling from a distribution and evaluating the model for each sample. It was shown that these repeated model evaluations consume great amounts of time and resources, making such methods infeasible in many cases. Also suffering from the implications of repeated model evaluations, SA was shown to be of importance when managing uncertainty in models. Emulation was introduced as a solution to this problem, building surrogate approximations of models to greatly accelerate evaluation time. However, both SA and emulation are complex, presenting a barrier to adoption.

The discussion on the Model Web outlined the concept, a SOA consisting of models and data exposed as Web services, as opposed to traditional local storage and execution, and highlighted the key benefits. Embracing SOA creates exciting opportunities for reuse, sharing, generic clients and tools, and the potential for building complex workflows from several simpler services. Interoperability was determined to be critical to the success of the Model Web. If interoperability is not achieved, model and data services and clients will be unable to communicate with one another, severely restricting the fundamental goals of the Model Web.

Web service technology designed to enable interoperability was discussed in 2.4, including those for services, and those for encoding and transferring data. Two sets of approaches for Web services were examined: those with no specific domain focus, and those within the geospatial domain. While these approaches share some similarities, such as the use of HTTP and XML, they differ in many other ways. The domain independent SOAP and WSDL standards have been adopted widely, and were found to be suitable for processing functionality, with the concrete service descriptions provided by WSDL enabling automatic service consumption, but lacking in metadata. The alternative approach in the geospatial domain developed by the OGC, realised as the WPS standard, was found to provide a good platform for exposing geospatial models, with standard support for asynchronous execution and metadata. However, the service descriptions provided by WPS lack the necessary detail to facilitate automated service consumption. The slow adoption rates for WPS were discussed, suggesting drawbacks to the specification, and insufficient tooling for clients. Further evidence of this was revealed with some authors resorting to the development of a WSDL proxy for WPS to increase usability. Considering these facts, it is unclear why the OGC have failed to fully embrace SOAP and WSDL technology for their service interfaces, and remains a topic for further research. REST architectures were briefly discussed as a simple approach to Web services, utilising HTTP methods more broadly than other technologies. While the principles of REST are clearly suited for data access services, it is unclear how they map to processing services, and are therefore excluded as a possibility.

Data encoding formats were discussed, and can help to ensure data can be transferred between heterogeneous services without translation. This discussion primarily involved GML and O&M for representing geometries and observations, and UncertML for probabilistic uncertainty. It was found that while these formats can enable interoperability, the abstract nature of O&M can lead to implementation challenges, and the specification technically allows anything to be used as the result of an observation. The growing popularity of JSON was discussed, with the simplicity of

the format making it a viable alternative to XML.

The discussion on Web-based workflows focussed on the technology for describing and executing complex geospatial workflows consisting of multiple model and data services. It was found that BPEL provides a rich vocabulary for describing such workflows, and is supported by execution engines, with many exposing a workflow as a reusable Web service. The language is however challenged by usability, as building the required BPEL document requires a degree of technical knowledge. Taverna Workbench, a GUI for the composition and execution of workflows, does not suffer from these usability problems, and is aimed at scientists rather than developers. Integrating geospatial models exposed as WPS with BPEL and Taverna workflows is a challenge due to the standard not supporting WSDL, again questioning why the OGC chose not to utilise the standard.

Uncertainty is clearly of great importance in modelling, and it is therefore surprising that the management of uncertainty within the Model Web remains an unexplored topic. Traditional desktop software tools are available for SA and emulation, but these require manual data transformation between model output and tool input. The potential of the Model Web for creating generic tools for SA and emulation is the primary motivation for the work of this thesis.

3

Models as Web services

CONTENTS

3.1	Foreword	64
3.2	Requirements for the Model Web	64
3.3	Web service interfaces	65
3.3.1	The WPS standard	65
3.3.2	Processing service framework	72
3.4	Model and interface integration	89
3.4.1	Execution patterns	90
3.4.2	Deployment challenges	92
3.4.3	Implementing communication	94
3.5	Summary	101

3.1 Foreword

Chapter 2 introduced the notion of sharing models and data over the Web, and concluded that interoperability and usability are critical to achieve this successfully. The following chapter discusses the implementation challenges of sharing models on the Web. Section 3.2 details requirements for the Model Web, and the features a suitable Web service interface is required to have.

Section 3.3 discusses interfaces for exposing models on the Web, with Section 3.3.1 containing a critical evaluation of a popular existing standard in the geospatial community, the WPS. An alternative solution developed as part of this work, the processing service framework, is introduced and evaluated in Section 3.3.2, aiming to solve many of the usability issues identified with the WPS.

Section 3.4 details the integration of Web service interfaces with existing models. Section 3.4.1 describes different model execution patterns, and some of the challenges associated with certain types of model. Additional challenges, unrelated to model execution characteristics, of exposing models on the Web are discussed in Section 3.4.2. Section 3.4.3 presents several implementation options to communicate with models from a Web service interface.

3.2 Requirements for the Model Web

The UncertWeb project is working to build the uncertainty-enabled Model Web, initially requiring the development of Model Web components. The choice of service interface for exposing models on the Web is critical, as both interoperability and usability must be provided. Within the context of the UncertWeb project, several desirable features were identified for a service interface exposing models and processes as components in the Model Web:

- Ease of use. The modellers who use the services exposing models may not be familiar with specifications and standards for Web technology.
- Client software and library support. With the previous point in mind, the barrier to use is much lower if existing software and libraries are available.
- Process development. Process development and deployment may be performed by the model owner who will not necessarily be familiar with Web service interface and data encoding standards. Support must also be provided for interaction with popular modelling languages; the Web service interface may be written in a different language.

- Workflow support. Building complex model workflows is a key benefit of the Model Web, making compatibility with associated standards and software important.
- Reference passing. Where geospatial data sets are extremely large, reference passing minimises data transferred between client and server.
- Asynchronous processing. With complex models and geospatial processes potentially taking hours to complete, asynchronous execution is desirable.
- Discovery and usage. Processes, inputs, and outputs must be sufficiently described to enable the potential for discovery and automated service consumption.

Although this list of requirements has been compiled specifically in the context of UncertWeb, the diversity of use cases in the project ensures they are relevant to a vast majority of scenarios in the Model Web. These requirements will form the basis of the assessment when considering a service interface for exposing models and processes on the Web.

3.3 Web service interfaces

3.3.1 The WPS standard

Widely accepted by the geospatial community (Brauner et al., 2009), but not necessarily widely used (Lopez-Pellicer et al., 2012), the WPS standard aims to expose geospatial processing functionality in an interoperable manner. When considering the interface features desirable for implementing the Model Web, the WPS meets several of the criteria. Any service implementing the standard must support asynchronous processing and reference passing, strongly supporting the Model Web by providing a guarantee that all WPS processes are compatible with these features. For discoverability and aiding consumption, properties of inputs and outputs can be described with additional metadata, including units of measure, supported CRSs, and a set of allowed values. To aid interoperability, the WPS follows a standard request and response message structure. This creates the potential for generic clients, as combined with the use of the `DescribeProcess` operation, they can build a request automatically.

The WPS also suffers from several drawbacks, critically affecting usability, client development, and workflow support. WPS processes are unrestricted in both input and output data type, and functionality. A WPS process can accept any data type, including XML-based and binary formats, and perform any operation on that data. Although this genericity allows the WPS to be

adopted in a wide variety of cases, such flexibility can also hinder interoperability. Each provider could implement processes using different data types, even where the data, or indeed process, are semantically equivalent. This could range from different versions of GML, to some providers forgoing GML completely and opting to use their own geospatial data representations. Client development and workflow integration involving such services thus requires additional translation and process specific code — something the WPS standard aims to minimise the need for.

The creation of application profiles is one possible solution to these interoperability challenges (Nash, 2008). A profile defines process inputs and outputs by name, and their associated data types. For example, a user could define a profile for a buffering process, where the inputs are GML geometries and a buffer width double value, and the output is the buffered geometry, again encoded as GML. Any implementing process must have these inputs and outputs to conform to the profile. By implementing support for this profile, a client can ensure all buffering processes are executable without additional code. The WPS profiling mechanism is appropriate in cases where a single type of process may have several different implementation algorithms. For the same operation, a user will be able to select a variety of processes, each potentially implementing a different algorithm to perform the process. The user, or client developer, only need support a single profile, rather than multiple processes.

However, there are many cases where the profiling mechanism is inadequate. The greatest usability challenge for services on the Web is not the identification of inputs or outputs, but the specification of data for those parameters. In the Model Web, the diversity of models means it is unlikely that inputs and outputs for two models will match, making the WPS profiling mechanism unadoptable. It is more likely however, that two models will use the same data types — be that a version of GML, O&M, UncertML, or another popular format. An alternative profiling mechanism could advertise a number of supported formats. For example, a ‘Model Web’ profile could include GML version 3.2, O&M version 2.0, and UncertML version 2.0. Therefore, a client developer can develop a client which can handle these three data formats, thus making it compatible with services which implement this profile.

To ensure services can be consumed, the genericity of the WPS must be supported by a strong description mechanism. The WPS provides a number of guarantees for a process consumer — the basic structure of an execution request and response, and a standard exception format. For every other aspect of the request, a consumer relies on the descriptions provided by the `DescribeProcess` operation. Without a process description, the consumer is unaware of the inputs they are

required to send, what outputs they can expect to receive, and all associated data formats. A strong description should provide enough detail for the client to consume the process, without ambiguity.

The current WPS standard version, 1.0, does not provide such descriptions. An XML data type described in a `DescribeProcess` response only provides a URL locating the schema the data must conform to. As an XML schema can, and typically will, contain a breadth of elements and types, merely specifying the schema URL is inadequate. A client will be unable to determine the exact data type to send, severely restricting usability and potential for GUI based software, automatic code generation, and workflow composition. If the client attempts to send data which validates against the schema, but is not supported by the process, the process can only return an exception to the client stating that the given type is unsupported, after which the client can select appropriate data and execute the process again.

Server-side implementations of the WPS specification have been developed in several languages, including Java (52°North WPS), Python (PyWPS) and C (ZOO Project). However, implementation in client software and tools is limited, consisting of a simple Java client library, plugins for uDig and QGIS, and support in the OpenLayers JavaScript mapping library. Due to the generic nature of the WPS, these tools can only support a restricted set of WPS processes. For example, the uDig plugin can send requests to a WPS, but it can only support a small number of input and output data formats. If a format is unsupported, the user is unable to execute the process, and an error message is returned by the plugin to the user. Process developers may consider WPS most suitable for implementing geospatial processes on the Web, contributing towards the popularity of the standard. However, important questions regarding usability, interoperability, and whether the standard actually excels in these characteristics, are scarcely asked.

Integration with existing technologies

The OGC decided not to adopt existing Web service protocols and description mechanisms when developing their specifications, causing the usage pattern of OGC Web services to differ from that of SOAP/WSDL services. For those who are not familiar with OGC Web services, this usage pattern may seem unnecessarily complex, and presents a barrier to adoption.

Finding and executing a process on a WPS firstly requires a call to the `GetCapabilities` operation. `GetCapabilities` is a common operation amongst all OGC Web service specifications, and is responsible for returning service information. Included in a `GetCapabilities` response

are contact details, supported protocols, available layers, or in the case of the WPS, available processes. To discover the information required to execute the process, a separate `DescribeProcess` request must be issued. The operation responds with a process description, containing a list of inputs and outputs, their associated data types and restrictions, and any additional metadata relating to the process. After retrieving the process description, the user has sufficient information to execute the process with an `Execute` request.

As `GetCapabilities` is standard across the OGC Web service stack (OGC 06-121r9, 2010), the usage pattern will be familiar to those within this domain. The `GetCapabilities` operation aims to provide consistency by returning standard metadata, in addition to service specific metadata. As every OGC service is guaranteed to respond to `GetCapabilities` in a similar manner, the use of the operation can aid service discovery. However, beyond standard metadata such as service keywords and provider contact details, the response only contains service specific information. For example, the WPS will return a list of process identifiers, whereas the WFS will return a list of feature types. This results in limited consistency across the service stack, and a minimal amount of reusable code.

By 2001, the SOAP and WSDL standards had been finalised by the W3C. With version 1.0 of the WPS specification not completed until 2007, it is unclear why a greater level of integration with W3C technologies was not considered by the OGC. For SOAP/WSDL services, a complete service description can be retrieved with a single request, which contains enough information to subsequently execute the process. The motivations are more unclear when considering the abundance of equivalent features in OGC and W3C specifications. For example, the SOAP fault element has an equivalent exception element in OGC specifications. To maximise reusability, the established SOAP fault element could have been used, as well as WSDL as an alternative to the `GetCapabilities/DescribeProcess` pattern. Instead of building new tools for OGC service specific elements, consumers could exploit existing libraries for SOAP and WSDL. The OGC could again argue the aim is to ensure consistency across their Web service stack, but it could be argued that when considering standards uptake, it is more important to provide consistency across the World Wide Web rather than a particular service stack.

If WSDL was adopted for WPS, issues relating to the lack of client implementations may not be so severe. As WSDL is widely supported, there is an array of available tools and compatible software. With Apache Axis and Microsoft Visual Studio, it is possible to quickly deploy a usable SOAP/WSDL Web service from existing code, or create a client for a service described with

WSDL. The client code generation makes it easy to integrate existing applications with Web services. Outside of code generation utilities, WSDL is also supported in workflow software such as Taverna and Kepler.

The mass-market approach of SOAP/WSDL services (Shade et al., 2012) has provided motivation for several efforts to integrate these standards with WPS. Ambiguity regarding such integration in version 1.0 of the WPS specification was partially resolved in OGC 08-009r1 (2008), which proposes two solutions based on providing a WSDL to either describe the whole WPS instance in a generic manner, or to individually describe each exposed process.

The most basic solution involves providing a generic WSDL document for any WPS instance. This approach is plagued by the abstract nature of the XML schema descriptions of process execution requests, drastically reducing the benefits of WSDL. Although the primary interface to a WPS is XML-based, the specification does not exploit the full potential of XML schema. While abstract schemas are available for WPS requests and responses, these schemas only specify that a process has a number of inputs and outputs with any data type.

If a model is exposed on a WPS, and subsequently included in a Taverna workflow, a WSDL document is required. With a generic WSDL document, the user is unaware of how many inputs are required, the identifiers and data types of these inputs, and what outputs they can expect to receive. The user must therefore manually build an `Execute` request document by augmenting the abstract message descriptions in the generic WSDL document with information gained as a result of the `DescribeProcess` operation. To understand how to interpret the `DescribeProcess` response in order to build process execution requests, the specification must be understood. These steps would not be necessary if concrete XML schema descriptions for each process were provided in the WSDL document. When a concrete WSDL document is imported, required inputs and data types are immediately identified, enabling Taverna to provide graphical interfaces for setting these inputs. It could be argued that XML schema cannot contain the same level of metadata as a `DescribeProcess` response, but several mechanisms exist for doing so, including the annotation element, or implementing a metadata retrieval operation on the service, essentially replicating the `DescribeProcess` operation.

When creating WSDL documents to individually describe each process exposed by a WPS instances, the request and response messages contained within the WSDL document will not validate against WPS schemas. This is due to the abstract nature of the WPS schemas, where it is impossible to add specificity to the `Input` and `Output` elements while maintaining compatibility

with the base schema. The benefits of using WPS are therefore reduced, as interoperability is lost as a result of non-conformance with the fixed message structures detailed in the specification. As there are no specification guidelines to follow, each service provider may have a different pattern for creating the WSDL messages.

Several solutions to automatically generate WSDL documents for WPS processes have been developed, either through proxies (Sancho-Jiménez et al., 2008), or as part of the underlying service implementation (de Jesus et al., 2011). However, these solutions have been found to generate inadequate message descriptions. For a service to be usable by third parties, a description must provide enough information for the user to determine the correct data type, or for a machine to utilise the description to discover appropriate data sources. We consider the message descriptions included in generated WSDL documents to be abstract, or weak. Listing 3.1 shows a comparison of different levels of abstraction in message descriptions. The first is the most abstract, as it permits any well-formed XML through the use of `anyType`. The second description is more concrete, as although the `Geometry` type is abstract itself, we now know that the element is any GML geometry — a technically feasible input in, for example, a buffering process. The final example shows a fully specified description which only permits one of three GML geometry elements.

```
<!-- Abstract description -->
<xsd:element name="Geometry" type="xsd:anyType"/>

<!-- More concrete description -->
<xsd:element name="Geometry" type="gml:Geometry"/>

<!-- Concrete description -->
<xsd:element name="Geometry">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="gml:Point"/>
      <xsd:element ref="gml:LineString"/>
      <xsd:element ref="gml:Polygon"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Listing 3.1: A comparison of abstract and concrete descriptions in XML schema.

Any WSDL generating solution for WPS services is limited to producing descriptions at the high level of abstraction seen in the first example. This limitation is a result of the current WPS description mechanism, as explained early in Section 3.3.1. For the process description to be usable, the developer can either provide additional metadata to specify the permitted elements from the schema, or subset the schema for each input and output, only including supported elements

and types. As the former is unstandardised, the latter option would be preferable. However, considering the large numbers of element and type combinations, this can require considerable implementation effort. This approach will also break clients relying on specific schema URLs to parse data, as these URLs will now locate schema subsets.

The clear benefits of concrete WSDL descriptions, and the inability to provide such descriptions for existing WPS instances prompted the development of an experimental proxy service. While the developed service is able to automatically produce WSDL descriptions for WPS instances in the same manner as other proxy solutions, the generated WSDL document contains concrete message descriptions. In order to create well-specified WSDL documents, additional metadata is required in a `DescribeProcess` response. Listing 3.2 shows an input description with metadata to specify that the input should be a GML point. The type is specified by a namespace URI and name, which must be defined by the schema referenced in the standard WPS `Format` element.

```
<Input minOccurs="1" maxOccurs="1">
  <ows:Identifier>Location</ows:Identifier>
  <ows:Title>Location</ows:Title>
  <ows:Metadata>@type="http://www.opengis.net/gml/3.2":Point</ows:Metadata>
  <ComplexData>
    <Default>
      <Format>
        <MimeType>text/XML</MimeType>
        <Schema>http://schemas.opengis.net/gml/3.2.1/gml.xsd</Schema>
      </Format>
    </Default>
    <Supported>
      <Format>
        <MimeType>text/XML</MimeType>
        <Schema>http://schemas.opengis.net/gml/3.2.1/gml.xsd</Schema>
      </Format>
    </Supported>
  </ComplexData>
</Input>
```

Listing 3.2: A description of an input with additional type annotation.

The proxy service generated WSDL documents usable in tools and software, with fully-specified input and output data types. Although minimal effort is required to annotate inputs and outputs in the `DescribeProcess` responses, it is necessary for each WPS process developer to add these annotations. As in many cases it will be the user, not the process developer, who needs the annotations, gaining process developer cooperation may be a challenge. Without standardisation through inclusion in the WPS specification, or acceptance as a best practice, others could create

similar, but not identical, annotations, thus limiting adoption and interoperability. The proxy is also an additional overhead, and where the hosting responsibility lies for the proxy, with either process consumer or developer, will likely be disputed.

Supporting current technologies is important in the rapidly evolving Web environment. Considering the growing interest in location, with smartphones and user content driven sites becoming location-aware, the matter is even more critical for geospatial communities. If current technologies are supported, more tools and support will be available, thus lowering the barrier to entry. Enforcing the use of common OGC Web service features restricts how services can be implemented. When new technologies surface, these features must be mapped to new paradigms, where it may not always straightforward, or even appropriate.

We can consider the recent example of JSON, a popular alternative format to XML for interchanging data. JSON is especially important for Web applications, where JavaScript is the client-side programming language of choice. As JSON is based on a subset of the JavaScript language, the format can be natively parsed into objects, and objects can be converted to JSON strings. In contrast, while XML in JavaScript can be natively parsed into the Document Object Model (DOM), the DOM must then be traversed to access data values. For developers of Web applications, the current lack of support for JSON in WPS could lead to longer implementation times.

3.3.2 Processing service framework

The issues with the WPS specification provided sufficient motivation to develop an alternative framework for exposing process and models on the Web. Enabling process consumption for a breadth of tools and clients was fundamental in this motivation, as well as the rapidly growing popularity of Web applications driving a requirement for JSON support. While many current WPS implementations assume the process developer has an understanding of XML encoding standards, such as GML and O&M, the framework should not make this assumption, and instead aims to simplify process development.

WSDL descriptions with concrete schema for all processes are automatically generated by the framework. A full specification of inputs and outputs is contained with the request and response elements for each process. Instead of merely referencing a schema URL, as seen in the WPS, the schema imports any schema URLs and makes a direct reference to the type, or abstract type, of the input or output data. In combination with WSDL, the primary interface of the framework

uses SOAP in an effort to maximise process consumption possibilities across a variety of tools and software.

The growing popularity of Web-based mapping libraries, such as Google Maps, OpenLayers and Leaflet, provided motivation for JSON support within the framework. JSON support is realised as an alternative interface to SOAP/WSDL, allowing a client to send requests and receive responses encoded as JSON, therefore completely bypassing XML. In contrast, the use of JSON in WPS 1.0 still requires XML parsing or encoding for all but the most simple of requests — a KVP encoded request for a process with a single output, returned as raw data.

Architecture

The framework is built on Java Servlet technology, which takes care of low-level HTTP handling, allowing us to directly access an input stream containing the request payload, and an output stream which the response payload can be written to. Building on Java Servlet technology enables the framework to be deployed on a number of Web application containers, including Apache Tomcat and Oracle GlassFish Server.

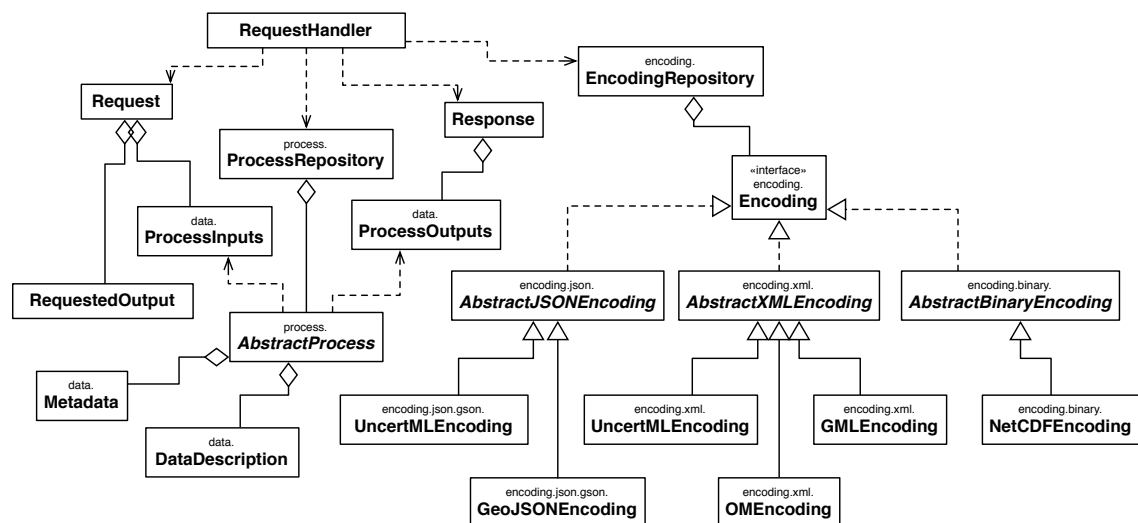


Figure 3.1: A simplified class diagram for the processing service framework.

Figure 3.1 shows the key classes in the framework, including those for request handling, encoding, and processing. The framework follows an extensible design, allowing developers to build processes, support additional encodings, and even implement new interfaces. After a HTTP request has been received by the Servlet, the request is forwarded to a handler for the appropriate interface, realised as a `RequestHandler` instance. This class is responsible for parsing the request data to an object, running the process, and encoding a response object. Two handlers, one for

Method signature	Return value
<code>getIdentifier()</code>	The unique process identifier.
<code>getMetadata()</code>	The process level metadata map.
<code>getInputIdentifiers()</code>	A list of input identifiers.
<code>getOutputIdentifiers()</code>	A list of output identifier.
<code>getInputDataDescription(identifier)</code>	The <code>DataDescription</code> associated with the given input identifier.
<code>getOutputDataDescription(identifier)</code>	The <code>DataDescription</code> associated with the given output identifier.
<code>getInputMetadata(identifier)</code>	The metadata map associated with the given input identifier.
<code>getOutputMetadata(identifier)</code>	The metadata map associated with the given output identifier.
<code>run(inputs)</code>	The outputs produced from processing the given set of inputs.

Table 3.1: A list of methods defined by the `AbstractProcess` class.

SOAP and one for JSON, are packaged with the framework, but a developer is free to implement additional interfaces if required. The request and response objects are decoupled from the interfaces, allowing a process or model developer to focus on the actual processing, rather than the Web service interface. This supports extensibility, as all existing processes will immediately be compatible with a new interface, and changes to existing interfaces will not affect the processes.

Process implementation

Processes are implemented on the framework by extending `AbstractProcess`, an abstract class defining several methods a process is required to have. These are listed in Table 3.1, and include methods for retrieving identifiers, input and output data descriptions, metadata, and performing the actual processing work. Extending a process class can allow developers to expose multiple versions of the same functionality without duplicating code. For example, a developer could provide one version of a process which exposes all parameters for users who wish to have more control, and another with a more limited set of parameters for users who require simplicity.

A data description contains a reference to the Java class of the input or output, and values specifying the minimum and maximum number of occurrences for that parameter in a request or response. If the minimum number of occurrences is zero, the parameter is treated as optional. The framework aims to make it easier to expose models on the Web, and helps to achieve this by only requiring a process developer to supply a reference to the desired Java type of the parameter, rather than specifying an encoding format in the data description. Using this type information, an

appropriate format is automatically selected by the framework — a benefit for those developers or model owners who may not be familiar with encoding standards. Conversely, developers who demand greater control over the encoding process can do so with custom encoding classes.

The `run` method is responsible for performing the processing task or model evaluation. When the method is called, it is passed an instance of `ProcessInputs` — a collection containing the process inputs contained within a request. Before the `run` method is called, inputs are parsed by the appropriate encoding classes, meaning the types of the objects contained match those specified by the associated `DataDescription` for an input. Individual inputs are retrieved from the collection by supplying their identifier. As an input can occur more than once in a request, it will either be retrieved as a single object, or a list containing every occurrence.

```
@Override
public ProcessOutputs run(ProcessInputs inputs) throws ProcessException {
    // get polygons and width inputs
    List<Polygon> polygons = inputs.get("Polygon").getAsMultipleInput().getObjectsAs(
        Polygon.class);
    double distance = inputs.get("Distance").getAsSingleInput().getObjectAs(Double.
        class);

    // perform buffering
    List<Polygon> bufferedPolygons = new ArrayList<Polygon>();
    for (Polygon polygon : polygons) {
        Polygon bufferedPolygon = (Polygon)polygon.buffer(distance);
        bufferedPolygons.add(bufferedPolygon);
    }

    // return output
    ProcessOutputs outputs = new ProcessOutputs();
    outputs.add(new MultipleOutput("BufferedPolygons", bufferedPolygons));
    return outputs;
}
```

Listing 3.3: The `run` method for a simple polygon buffering process.

Once the inputs are retrieved, the developer is free to implement the process as they require. A wealth of implementation options are available, including utilising third-party libraries to perform calculations, importing existing code, or communicating with an external program or model. Listing 3.3 shows the `run` method for a simple polygon buffering process, which utilises the JTS Topology Suite (JTS) library to create a buffer region around a list of polygons. Approaches to model communication are discussed further in Section 3.4. Process outputs are handled in a similar manner to the inputs, by means of a `ProcessOutputs` object, which the developer is responsible for instantiating and adding outputs to. An output can either be a single object or a list, depending again on the number of occurrences specified in the `DataDescription`. The outputs

collection must be returned by the `run` method, after which control is returned to the framework handler classes.

Implementing the three metadata retrieval methods — `getMetadata`, `getInputMetadata` and `getOutputMetadata` — is optional, as either of these can return a null value if the process developer does not wish to provide metadata. However, metadata is essential within the Model Web, and can drastically improve process discovery, consumption and workflow composition, in both manual and automated contexts. With the absence of metadata, discovery queries are restricted to the identifiers of processes, inputs and outputs, and any data type information contained within a service description document. In the case of automated or semi-automated workflow composition, this information will rarely be sufficient to discover other processes and appropriate input data sources.

The framework implements metadata in the form of tag based annotations, where each unique piece of metadata has a tag name defining the property the value relates to. Originally designed to assist users during workflow composition, the annotations are both human and machine readable, presenting the opportunity for automated workflow composition. If enough annotations are provided, it is technically possible for a workflow composition tool to parse the tag values, find an appropriate data source, and automatically specify interactions between services. A dictionary of metadata tags was developed within the UncertWeb project, allowing a user to specify various spatial, temporal and variable properties for a model and its parameters (Jones et al., 2012). Example tags include:

spatial-crss to specify the supported spatial reference systems;

temporal-resolutions to indicate the temporal support of the variable values;

variable-uncertainty-types the UncertML type used to describe the variable uncertainty.

A dictionary such as this is imperative in the success of the Model Web to ensure model owners across various domains and institutions share common definitions. Use of the UncertWeb dictionary combined with the framework replicates standard WPS metadata, including supported CRSs and units of measure for a variable, and does so in an extensible manner, allowing new tags to be defined independently of the service implementation.

Each process class is self-describing — they are guaranteed to return a list of inputs, outputs, and associated data descriptions, thus enabling the framework to automatically generate WSDL

and XML schema documents. For a process, the XML schema generator will find suitable encoding classes for each process input and output, based on the Java class specified in their DataDescription. If an XML encoding class can be found, the schema generator will import the schema for the encoding format, and retrieve the name of the XML element or type in that schema which maps to the Java class. An example schema document containing a request element for a buffer process is shown in Listing 3.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ps="http://
www.uncertweb.org/ProcessingService" xmlns:gml="http://www.opengis.net/gml/3.2"
targetNamespace="http://www.uncertweb.org/ProcessingService" elementFormDefault="
qualified">
  <xsd:import namespace="http://www.opengis.net/gml/3.2" schemaLocation="http://52
north.org/schema/geostatistics/uncertweb/Profiles/GML/UncertWeb_GML.xsd" />
  <!-- DataReference element omitted -->
  <xsd:element name="PolygonBufferRequest">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Polygon" minOccurs="1" maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>
              @description The polygon(s) to buffer
              @spatial-crss epsg:4326, epsg:27700
            </xsd:documentation>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:choice>
              <xsd:element ref="gml:Polygon" />
              <xsd:element ref="ps:DataReference" />
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Distance" minOccurs="1" maxOccurs="1" type="xsd:double">
          <xsd:annotation>
            <xsd:documentation>
              @description The distance to buffer the polygon(s)
              @variable-units-of-measure metres
            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <!-- Response omitted -->
</xsd:schema>
```

Listing 3.4: The generated schema for a buffer process XML schema.

If specified by the process developer, metadata is added to the generated schema document. XML schema contains an annotation element, which can be placed within another element to provide comments. Where metadata is available for an input or output, an annotation is placed within

the generated element for that parameter. A documentation element is added to the annotation, where metadata is placed. In the style of Javadoc annotations, each metadata tag is preceded by the at sign, after which a space is inserted before the value of the property.

The adoption of XML schema for message descriptions enables each request made to the SOAP interface to be validated, ensuring that the encoding classes receive both well-formed and valid XML elements to be subsequently parsed. This drastically reduced the need for error checking in the SOAP handler class. While the error messages generated through XML validation are concise, they lack user-friendliness, and therefore require additional translation before returning to the user.

Although there are no direct WSDL equivalents available in JSON, technologies are available for describing JSON-based services and data, including the Google API Discovery Document¹ and JSON schema². As both of these require significant implementation effort, and lack both server and client-side libraries, an alternative approach was adopted. A JSON formatted document is generated by the framework, containing an array of objects describing each process. These objects contain the identifier of the process, and input and output descriptions. Each parameter description contains an identifier, a type, multiplicity, and any metadata tags supplied by the process developer. While the approach is non-standardised, it does expose the potential for automatic service discovery and consumption. However, the approach is somewhat limited by the weak type descriptions, which simply provide a Java class name rather than machine-readable schema describing the structure and properties the data must contain.

Data encoding

As with the interface classes, encoding is decoupled from the request, response, and processing classes, negating any requirement for the process developer to be familiar with XML, JSON, or any Web service interface standards. Three types of encoding are supported by default — XML, JSON, and binary, but additional types can be added by extending the `AbstractEncoding` class. All encoding classes, independent of format, are added to a repository during service initialisation. The repository can be queried for an encoding based on Java class, MIME type, or format.

An XML encoding class must implement a number of methods. For convenience, subclasses of `AbstractXMLEncoding` can either deal directly with input and output streams, or with JDOM objects. JDOM objects enable easy navigation through an XML document, but may not be the

¹<https://developers.google.com/discovery/v1/reference/apis>

²<http://json-schema.org/>

Method signature	Return value
<code>getNamespace()</code>	The namespace of elements in the schema.
<code>getSchemaLocation()</code>	The URL locating the schema for this encoding.
<code>isSupportedType(type)</code>	A boolean value indicating whether the given Java type is supported by this encoding class.
<code>getInclude(type)</code>	An element name from the schema which maps to the given Java type.
<code>parse(element, type)</code>	An object, which is an instance of the given type, parsed from the XML element.
<code>encode(object)</code>	An XML element representing the given object.

Table 3.2: A list of methods defined by the `AbstractXMLEncoding` class.

preferred library of choice for some developers.

To enable use in a variety of cases, the framework supports GML, O&M, and UncertML. As GML and O&M contain many abstract elements, implementing fully-featured parsers for the base schemas is impossible, as well as restricting specificity of the generated schemas. Consider the use of O&M, where an observation can have a result of any type. For this reason, the profiling of these abstract schemas, where elements in the schema are restricted, is encouraged (Lake, 2005). Profiles for both GML and O&M were developed within the UncertWeb project, with the aim of supporting the various use cases. The GML profile restricts geometry types to primitives — points, lines, polygons and grids. Profiling is even more critical for O&M, where the result is by default, completely unrestricted in type. In the UncertWeb O&M profile, the base observation type has been extended to provide a new set of observation types with restricted result types. For example, the result of the `OM_UncertaintyObservation` type can only contain UncertML, and the result of the `OM_TextObservation` type can only contain string values.

The creation of profiles for GML and O&M made it possible to create a fully-featured Java APIs for these formats, capable of parsing and generating XML, and representing elements as objects. Where existing libraries are available for parsing and generating data in the desired format, such as the UncertWeb GML and O&M profiles, the encoding class for the framework is simply a wrapper around these libraries. This can make supporting an additional encoding format extremely fast, as the majority of effort required to build encoding classes is in the serialisation and deserialisation code.

JSON data encoding is performed by the Google Gson library³, which will attempt to automatically parse and encode the data. There may be cases where Gson is unable to automatically parse or encode data types, or where more control is required, for example, if a field in the Java

³<https://code.google.com/p/google-gson/>

object should be mapped to a field in the JSON object with a different name. In such situations, custom Gson serialisation and deserialisation classes can be created, several of which are included in the framework for geometry, O&M, and UncertML types.

```
{ "PolygonBufferRequest": {
  "Polygon": [
    { "type": "Polygon",
      "coordinates": [[216521.28107019, 771211.422979205], [216601.11240104,
787001.312678018]],
      "crs": {
        "type": "name",
        "properties": {
          "name": "http://www.opengis.net/def/crs/EPSG/0/27700"
        }
      }
    }
  ],
  "DataReference": { "href": "http://some.url/anotherpolygon.json", "mimeType"
: "application/json" }
},
  "Distance": 100.0
} }
```

Listing 3.5: A JSON encoded request for a buffering process.

While geometry objects are encoded as GML in the SOAP interface, the OGC have yet to define a JSON encoding for GML. However, the GeoJSON format can represent geometries and features as JSON objects, and shares much in common with GML. Although yet to be approved for inclusion in the O&M standard, a JSON encoding for O&M has been developed with the UncertWeb project, and supported by the O&M profile API. Listing 3.5 shows an example JSON request for a buffering process. The flexibility of JSON permits inputs with multiple occurrences to be sent as an array, as can be seen in the array of polygons, where the example includes one polygon encoded inline as GeoJSON, and the other as a reference.

When large geospatial datasets are used, embedding input data in a request is often infeasible. As an alternative to inline data, a processing request made to the framework can supply a Web-accessible URL locating the data. The framework is responsible for resolving the URL and downloading the data, which is then parsed in the same manner as inline data. The process developer need not implement special support for referenced data, as the inputs object sent to the `run` method is equivalent regardless of whether the data was supplied inline or as a reference. This approach can drastically reduce data transfer between client and service, especially in scenarios where data is published to a Web service, allowing the processing service to communicate directly with the data service.

A process consumer can also request the outputs be returned as referenced data. As with input

references, the process developer is not required to implement extra provisions for referenced outputs, as the framework is responsible for storing the data and generating a Web-accessible URL. Output data references may be especially useful in workflows, as the reference URL can be sent as an input reference to another process, therefore eliminating any need for the workflow orchestrator to download the data before sending to the next service.

Some geospatial data types, such as raster maps, are extremely inefficient to represent in text-based formats. Binary formats are supported by the framework using the aforementioned data reference mechanisms. However, as binary data cannot be described by schema as with XML, the MIME type of the format must instead be given. The MIME type is used to find a suitable binary encoding class from the process repository. In contrast to an XML encoding class, those for binary formats are less complex, as a binary encoding class need only implement methods for parsing and encoding data, and one to indicate whether a given MIME type is supported. When parsing data, a binary encoding class is passed the input stream, from which it must return an appropriate object send to the `run` method of the process. Instead of loading the data into memory, it may be more efficient for the class to write data to a file, and pass an object to the process which is capable of streaming from the file. For encoding binary data, the encoding class must write the output object to a stream.

Service deployment

The minimal steps required for implementing the framework are to import the library, create a deployment descriptor, and configuration file. A deployment descriptor is essential for the Web application container, as it describes how Servlet classes are mapped to a set of URLs. The configuration file contains the fully-qualified package and class names of any additional process and encoding classes the framework should load on initialisation. For encoding classes supporting the same Java types, those specified in the configuration have priority over those provided with the framework. Additional environment properties can also be specified in the configuration file, which will then be available throughout the framework.

Client usage

A major difference between the framework and the WPS is full support for SOAP, WSDL, XML schema, and JSON. These technologies have been implemented with the aim of providing better support for client development in standalone tools, Web applications, and workflow software. To

provide a basic means of testing usability and support in these areas, an example service built with the framework was developed. The service contains three processes: one to buffer polygons, one to perform polygon intersection, and another to measure the area of a polygon. These three services can form part of a contrived workflow, for example, to measure the area of a country effected by a disaster, such as a forest fire. For comparison, these processes were also exposed on a WPS, using version 2.0 RC8 of the 52°North WPS implementation.

The adoption of WSDL and XML schema enables the automatic generation of client-side code for services. A variety of code generation tools are available for many languages, including Visual Studio for C#⁴, SUDS⁵ for Python, and soap4r⁶ for Ruby. Client-side code usually consists of a service client and data binding classes, which map XML schema elements to native language classes, allowing a user to execute a process on a Web service without directly handling any XML. The usability of code generated with these tools is dependant on the quality of the source WSDL documents, with weakly specified documents resulting in abstract code. Abstract code is typically difficult to use, as the user of the code may not have enough information to supply the correct data to an input, or to implement correct output handling.

To evaluate the usability of code generated using WSDL documents provided by the framework, Apache Axis⁷, a Java platform for creating Web service applications, was used to generate client-side code. Listing 3.6 shows a simple example of the generated code being used to buffer a polygon, passed to the process by data reference. The classes produced by Apache Axis allow a developer to discover which inputs can be set in the request, and what data type they are required to be.

When using simple data types, for example the polygon buffer distance, setting an input is as simple as calling a method on the request object, passing a Java primitive value. Where complex data types are used, for example the polygon to buffer, it is possible to either supply a data reference, or build a polygon with the generated data binding classes. As the binding classes are merely a Java class representation of XML elements, the usability of these binding classes decreases as schema complexity increases. In situations where schema complexity is high and data reference passing is not possible, it may be preferable to utilise a library to parse and encode data.

While generated code is not suitable for all cases, it can be especially useful for rapid client development, testing, and integration with existing applications. If functionality in an application

⁴<http://www.microsoft.com/visualstudio/eng>

⁵<https://fedorahosted.org/suds/>

⁶<https://github.com/rubyjedi/soap4r>

⁷<http://axis.apache.org/axis/>

```
// create the client stub
ProcessingServiceStub stub = new ProcessingServiceStub();

// create request
PolygonBufferRequestDocument polygonBufferRequestDoc = PolygonBufferRequestDocument
.Factory.newInstance();
PolygonBufferRequest polygonBufferRequest = polygonBufferRequestDoc.
addNewPolygonBufferRequest();

// add polygon reference
DataReference ref = polygonBufferRequest.addNewPolygon().addNewDataReference();
ref.setHref("http://uncertws.aston.ac.uk/data/tasmania_polygon_2.xml");
ref.setMimeType("text/xml");

// set distance
polygonBufferRequest.setDistance(1.0);

// send request
PolygonBufferResponseDocument polygonBufferResponseDoc = null;
try {
    polygonBufferResponseDoc = stub.polygonBuffer(polygonBufferRequestDoc);
    PolygonBufferResponse polygonBufferResponse = polygonBufferResponseDoc.
getPolygonBufferResponse();

    // get the buffered polygons
    BufferedPolygon[] bufferedPolygons = polygonBufferResponse.
getBufferedPolygonArray();
}
catch (RemoteException e) {
    // print error message
    System.err.println(e.getMessage());
}
```

Listing 3.6: Executing the polygon buffer process using code generated by Apache Axis.

is currently performed by a call to a local method, it may be as simple as to switch that method with a call to the generated client.

When services are known before compile time, code generation is a rapid and straightforward Web service consumption method. However, there may be cases where the addition, modification, and removal of services will be required after compilation, such as that of the generic client, where the aim is to support all services conforming to a predefined standard or message pattern. In a similar manner to the WPS, the framework adopts a fixed message pattern for requests and responses, independent of process implementation. A fixed message pattern creates the possibility to build generic tools.

Any process on the framework can be executed from browser JavaScript clients using the JSON interface. Building a request is as simple as creating a JavaScript object, which can then be serialised to JSON using the built-in support included in all major browsers. Sending a request will commonly be performed using Asynchronous JavaScript and XML (AJAX), with libraries such as jQuery capable of parsing the received JSON to native a JavaScript object. The use of GeoJSON allows results to be easily displayed using popular Web mapping libraries. Listing 3.7 shows the polygon buffering process being called from JavaScript; note how the request object is constructed as a native object.

The non-standard nature of the JSON process descriptions provided by the framework mean client-side code generating libraries are not available as they are for WSDL documents. However, the descriptions can still be used to create generic clients, capable of automatically parsing a description and providing appropriate input options to the user. This technique has been adopted by the emulator running service, which is described in detail in Chapter 4.

Browser implementations of JavaScript adhere to a strict same-origin policy, where scripts are prevented from performing most AJAX requests unless the requested resource is from the same origin. The origin is defined as a combination of protocol, hostname, and port number, and examples of allowed and disallowed are listed in Table 3.3. This can severely limit JavaScript client development in the Model Web, where we may frequently wish to use models and data from external providers. The problem may also exist when working internally, as services will often be available on different ports to that of the client.

These restrictions can be overcome with the use of cross-origin resource sharing (CORS), a mechanism that allows scripts to use AJAX to access resources from different origins. CORS adds new HTTP headers that allow servers to specify which domains they permit resources to be

```

// load polygon asynchronously
var polygon;
var polygonXHR = $.get('tasmania_polygon_2.json', function(data) {
    polygon = data;
});

// when polygon is loaded
$.when(polygonXHR)
    .then(function() {
        // create request
        var bufferRequest = { PolygonBufferRequest: {
            Polygon: [ polygon ],
            Distance: 1.0
        } };

        // send request asynchronously
        $.post('http://localhost:8080/ps-example/service/json', JSON.stringify(
bufferRequest), function(data) {
            // get result
            var bufferedPolygon = data.PolygonBufferResponse.BufferedPolygon[0];

            // add to Leaflet map
            L.geoJson(bufferedPolygon).addTo(map);
        });
    });
});

```

Listing 3.7: Buffering a polygon in JavaScript using the JSON interface.

Resource	Allowed?	Reason
http://www.uncertweb.org/data	Yes.	Same protocol, hostname, and port.
http://www.uncertml.org/data	No.	Different hostname.
ftp://www.uncertweb.org/data	No.	Different protocol.
http://www.uncertweb.org:8080/data	No.	Different port.

Table 3.3: Allowed and disallowed HTTP requests called from origin <http://www.uncertweb.org>.

served to. These headers are supported by browsers, which then enforce restrictions defined by the server. Web services exposing models will typically grant access to all origins, and as such, the processing service framework can be CORS enabled.

Workflow usage

To test the usability of the framework in workflow software and languages, a basic workflow scenario was created. This scenario involves buffering a distance around Cape Barren Island in Tasmania, performing an intersection with the island of Tasmania, and finally measuring the resulting area of the intersection. Although highly simplified, this workflow could provide the basis for analysing the local impact of a natural disaster on the Cape Barren Island. The workflow was implemented in both Taverna Workbench and BPEL.

Taverna allows graphical composition and orchestration of workflows, and has built-in support for WSDL. Adding a service simply requires one to give a URL locating the WSDL file, after which Taverna will import processes described by the document into the service panel. In addition to remote, Web-based services in the panel, Taverna allows workflows to include local services, such as fragments of Java code, and interaction with R. The example workflow will however only include the three processes exposed by our Web service.

When dealing with complex XML inputs, Taverna provides a user-friendly XML splitter mechanism as an alternative to XML query languages, such as XPath. The XML splitter displays elements within a complex type as separate inputs in the composition window, allowing values to be assigned to individual properties, rather than having to specify the whole element as XML. This could be especially useful to users who are not familiar with XML encoding standards, but may still be challenging for data formats with a large number of properties, such as O&M. Once splitters are created for process request and response documents, we can match the output from the buffering process to an input on the intersection process, and the output from the intersection process to the input of the area process.

The URLs locating the polygon of the hazard source and the polygon of the potentially affected site are defined as workflow inputs, and the affected site polygon and area are defined as outputs. It would be possible to supply inline data for the inputs, but requiring URLs enables data to be retrieved from other Web services, thus further supporting goals of the Model Web. Figure 3.2 shows the final workflow scenario composition in Taverna. Workflow components shown between services, for example 'Polygon_DataReference', are XML splitters which have been created using

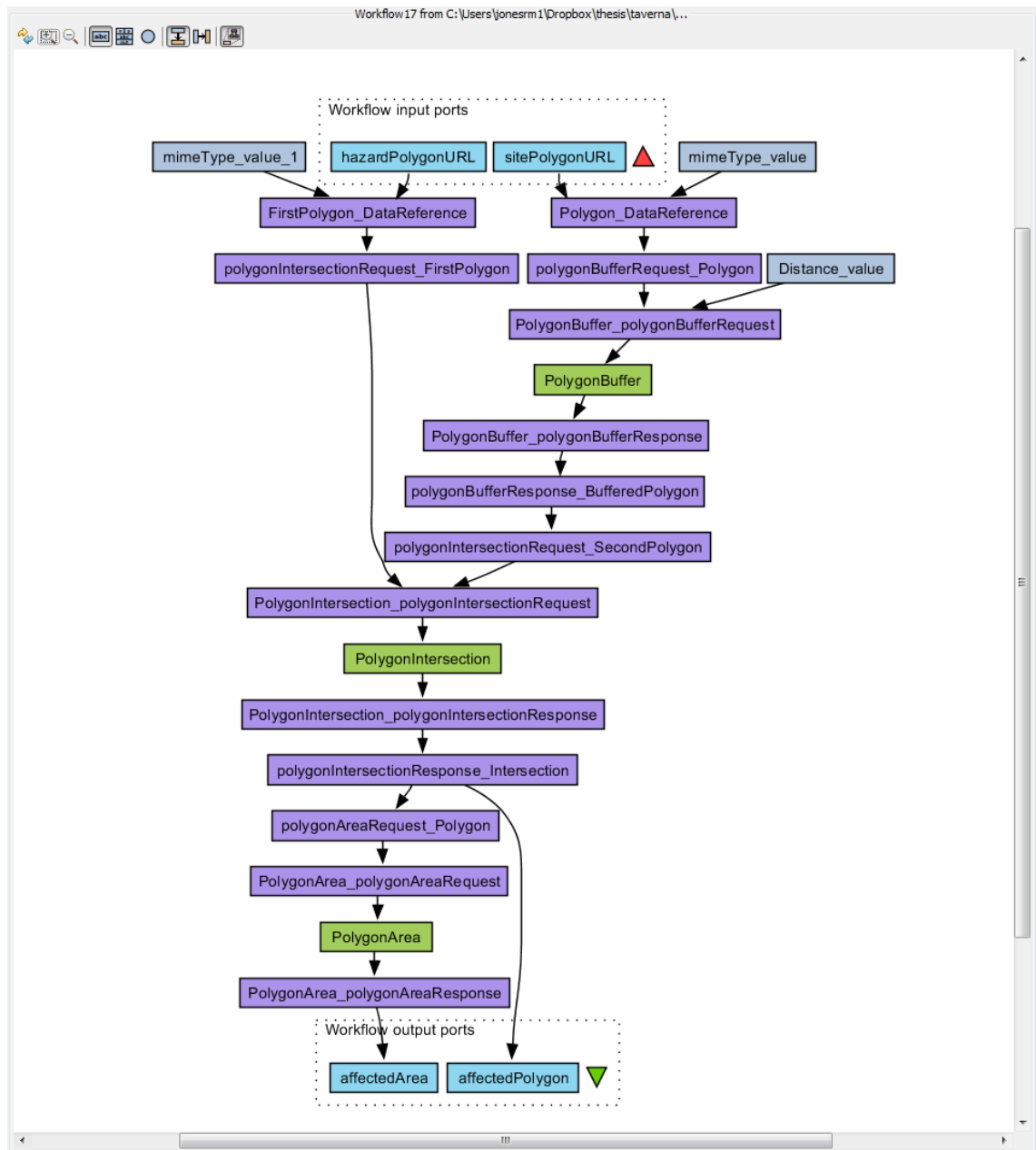


Figure 3.2: The workflow scenario composed in Taverna with processes exposed using the framework.

the graphical interface. The result of the intersection process, the affected area, and the calculated size of that area are defined as workflow outputs.

WPS integration with Taverna is by contrast, more difficult. A generic WSDL document describing any WPS does not contain any process specific input and output information, meaning Taverna is unable to employ the XML splitter mechanism. A user must instead deal with XML themselves to build the required requests, and parse response documents. These problems have been alleviated by several automatic WSDL generating approaches for WPS (Sancho-Jiménez et al., 2008; de Jesus et al., 2011), but are currently non-standardised, and often service implementation specific.

A clear limitation of Taverna in all examples is the lack of built-in support for geospatial data, including data import and visualisation. However, Taverna does allow outputs to be saved, which could then be opened using appropriate geospatial software. Additionally, in a completely SOA approach, any geospatial outputs could be saved on a data service, such as a WFS, after which they could be retrieved as images and displayed within the Taverna Workbench.

BPEL is a workflow definition language, and relies on tools to support its use. Those tools which are available, such as the Eclipse BPEL Designer, are significantly less-user friendly than Taverna. While BPEL can be integrated to provide workflow serialisation in graphical tools similar to Taverna, BPEL may be more suitable for automated composition mechanisms, where the user has no direct control over the workflow.

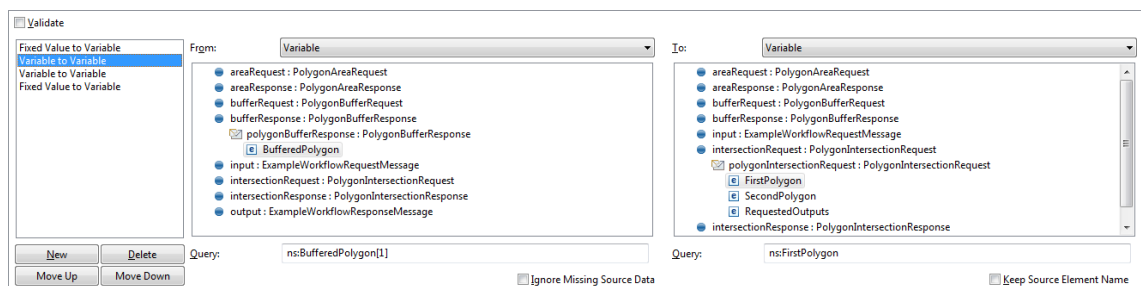


Figure 3.3: Input and output matching in Eclipse BPEL Designer between processes exposed using the framework.

A BPEL document for the example workflow was built using the Eclipse BPEL Designer. Although creating the workflow with BPEL is more complex than Taverna, the use of WSDL and concrete XML schema does ease the process, as the BPEL Designer provides the ability to match inputs and outputs. This is demonstrated in Figure 3.3, which shows the polygon buffer output being assigned to the first polygon intersection input. XPath is a query language for XML documents, which allows elements, attributes and values in an XML value to be located by expression,

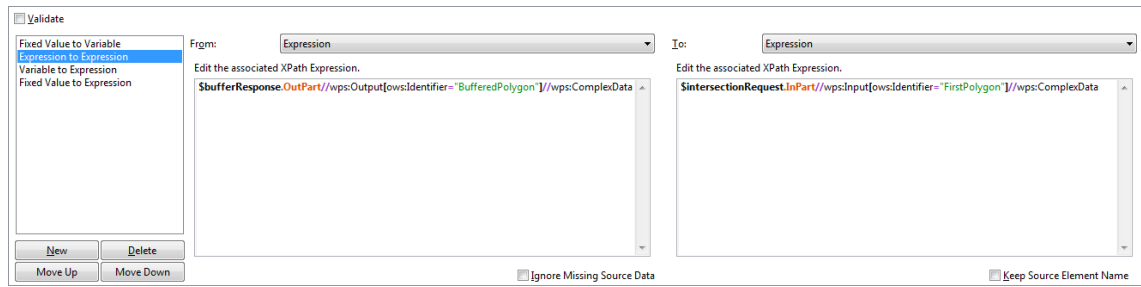


Figure 3.4: Input and output matching in Eclipse BPEL Designer between processes exposed on a WPS.

and is utilised by BPEL to copy values to and from request and response messages. When inputs and outputs are matched, the BPEL Designer automatically creates the required XPath expression, to specify that an output from one process should be set as an input to another process.

Figure 3.4 shows the same input and output matching GUI in Figure 3.3, but for processes exposed on a WPS. As with Taverna, the lack of real support for WSDL combined with weak schema documents means the tool is not provided with sufficient information to allow selection of inputs and outputs in the same way as when using processes exposed on the framework. Therefore, a user must manually build XPath expressions to locate the desired inputs and outputs, demonstrating the additional effort required to include WPS processes in BPEL workflows.

The workflow was deployed on the Apache ODE orchestration engine. A key benefit of BPEL is the use of such engines, which automatically expose the finished workflow as a standard SOAP/WSDL Web service. Client-side code for executing the workflow was generated using Apache Axis, in the same manner utilised for the WSDL documents offered by the framework.

3.4 Model and interface integration

When tasked with the development of the Model Web, selecting a suitable Web service interface enables interoperability, simplifying consumption by clients and other services (Bieberstein et al., 2005), and improving workflows (Zhao et al., 2012). However, the Web service interface is only part of the picture, as the interface must evaluate an underlying model. This integration of model and interface will typically involve an additional communication layer, responsible for converting inputs to model-compatible formats, evaluating the model, and converting outputs into Web-compatible formats.

Although often taken for granted, communicating with a model from a Web service interface can involve significant implementation effort, and raises an important question in where the re-

sponsibility for that effort belongs. Model owners and users should be aware of how the model functions, how inputs are specified, and how outputs are returned, but may know little about Web service interfaces. In contrast, process or Web service developers are typically experts on how to expose functionality on the Web, but may be unaware of the functionality of a model. While the framework discussed in Section 3.3.2 attempts to resolve this issue by simplifying process development, there are still a number of challenges faced by those exposing models on the Web.

3.4.1 Execution patterns

The abstract term ‘model’, can apply to any function that returns an output for a given input, therefore allowing models to be developed in any language. Languages commonly used to implement models range from those which compile to machine code (C, Fortran), those which compile to bytecode (Java), and interpreted languages (MATLAB and R). Additionally, a model can be developed with a standard interface, ensuring that all conforming models can be executed in the same way. The choice of language, means of interaction (graphical or command-line interface), and adoption of a standard interface will strongly influence how a model can be executed by a Web service interface. This section discusses execution patterns for compiled and interpreted language models, and those conforming to a standard interface.

Compiled language models

When communicating with a compiled language model, we rely on the model to be written in a manner that is suitable for automated execution. As we are only dealing with request-response Web service interaction, the model must not require any additional decisions to be made during runtime. Inputs must therefore be supplied either as execution parameters, files readable by the model, or through another source, such as a database. Outputs must be returned through standard system output, files on disk, or saved to other data sources. The major role of the Web service interface for such models is handling input and output data, ensuring that this is done such a manner to successfully execute the model and return the correct outputs.

This pattern for process to model communication immediately excludes models which are only accessible through a GUI application. While it may be technically feasible to script mouse movement and key presses in order to automate GUI model execution, there are several impracticalities to this approach. As key presses are only sent to the window with focus, subsequent key presses will be sent to the wrong window if focus is lost during communication, therefore caus-

ing the process to fail. Unless export options are provided, capturing output and messages from the GUI is infeasible, and can involve screen capture combined with optical character recognition (OCR). Creating scripted sequences for mouse movement and key presses typically requires specialist, platform-specific, software. The use of such software adds a layer of complexity to model communication, and involves a great deal of additional implementation effort.

If source code is available for the model, it may be possible to remove the GUI to enable automated execution of the model. However, source code may be difficult to acquire for many models, and especially those where development has been inactive for a prolonged period of time. Even once source code is obtained, the code must be understood sufficiently for the necessary modifications to be performed without harming the model itself. The difficulties involved in source code modification are accelerated in cases where the code is undocumented, or if the GUI is tightly coupled to the model.

Interpreted language models

For interpreted language models, the Web service interface must communicate with the software interpreter, rather than any compiled executables. This communication could be as simple as executing a command line interpreter on the model code, specifying inputs as arguments, and parsing the data from standard output. In a service scenario, and especially one where uncertainty analysis is performed through multiple model evaluations, the overhead of starting the interpreter for each mode evaluation may be significant. This overhead increases when the interpreter initialises an environment, such as those of a numerical nature seen in MATLAB and R.

Program execution restrictions set by operating systems can make specifying high-dimensional model inputs as command line arguments impossible. Consequently, it may be more appropriate for interpreted language models to be programmed with support for file-based inputs and outputs, or handling of data from other sources. For such models, the Web service interface will be similar to those developed for compiled language models, and possess the main responsibility of handling input and output files. However, unless a model already provides support for file-based input and output, execution in this manner will require changes to the model code.

It is more common for models executable in numerical environments to simply be functions that are called, passing existing variables as parameters. For such models, the Web service interface must employ a mechanism to communicate with the environment directly, enabling the interface to assign values to variables, call the desired model function, and retrieve output variables.

To eliminate the overhead introduced by initialising the environment for each model execution, it may be preferable to instead start the environment prior to model evaluation, which the service interface can then communicate with.

Standard modelling interfaces

While SOAP/WSDL and OGC Web Services (OWS) aim to provide interoperability between Web services, modelling interfaces, such as Open Modelling Interface (OpenMI), seek to do the same for models (Gregersen et al., 2007). These interfaces can be adopted in either compiled or interpreted languages, and ensure that all models, inputs, and outputs can be described in the same manner, enabling data exchange between multidisciplinary models and software components. Using a modelling interface, a model developer must implement the defined methods to conform to the specification. For OpenMI, a library has been developed in C# and Java, containing a set of interfaces and classes to support this, and shares many similarities to the `AbstractProcess` class in the processing service framework. An implementing model class can then be compiled, if required, to form components usable by software. OpenMI develop Pipistrelle, a GUI based application for creating compositions of OpenMI components.

Modelling interfaces and the Model Web concept share much in common, as they both aim to achieve interoperability between models. However, many modelling interfaces, including OpenMI, are only designed for local models, therefore provide fewer opportunities for discovery, sharing, and use compared to the Model Web. Instead of considering OpenMI as an alternative to accessing models through Web services, it can be integrated into Web service components, reducing the amount of communication implementation, as all models can be guaranteed to behave in the same way.

3.4.2 Deployment challenges

Independent of language choice, several challenges are faced when deploying models on the Web. These issues include scalability issues caused by increased model access, model input and output data encoding formats, and target platform compatibility.

Models are often computationally expensive, evaluating complex functions and dealing with large datasets. In production environments, a Web service container could be running a number of applications, not only limited to those for modelling. Under constant demand, performance of Web services hosted on the same machine as the models could suffer drastically. A decrease in

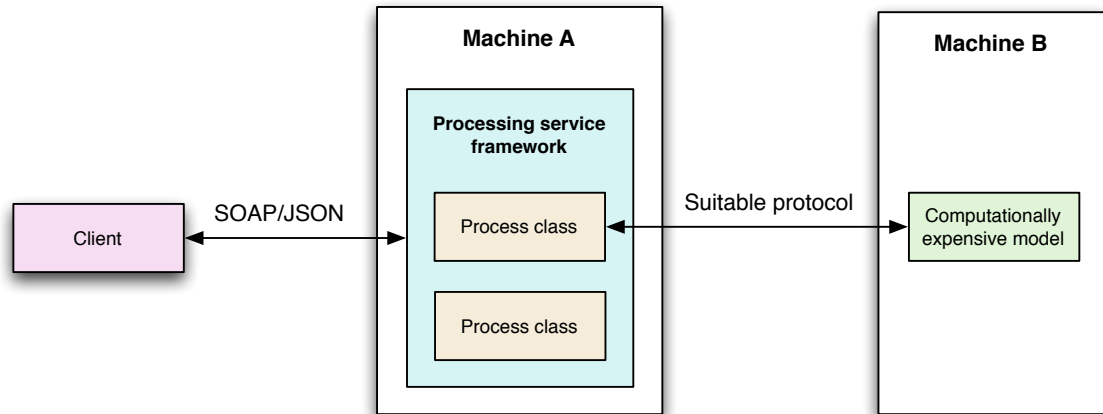


Figure 3.5: Communicating with a model on a separate machine to the Web service.

performance is unlikely to be acceptable, especially where services are critical. It may therefore be more feasible to offload model execution to a separate machine, as shown in Figure 3.5, ensuring the best possible Web service performance during model evaluation. The ability to remotely communicate in such a manner is unlikely to have been considered at the time of model development, thus adding to the implementation effort for those tasked with exposing models through Web service interfaces.

In general, models do not encode inputs or outputs in a standard manner. Especially when a GUI interface is provided, many model developers will not consider the potential for input files to be generated by external mechanisms. Data is typically encoded as plain text, including comma-separated values (CSV) and tab-separated values (TSV). The use of non-standard data formats requires the Web service interface to undertake additional data transformation steps. Inputs received by the interface will be objects representing elements in, for example, GML and O&M, and thus are converted by the process class into a format understood by the model. After model evaluation, the reverse of this step is performed, where outputs are transformed into objects representing elements defined by suitable data formats.

A requirement of the Model Web is platform independence, aiming to provide universal access to models and data sources. While the platform neutral SOAP and JSON service interfaces provided by the processing service framework satisfy this requirement, the models we wish to communicate with may only be compatible with specific platforms (i.e. operating system, architecture). The ability to communicate with the model from the Web service interface therefore depends on the platform where the service is deployed. If the deployment platform is not compatible with the model, we must either switch platforms, recompile the model from source and targeting the desired platform, or utilise a compatibility layer.

Switching platforms is unlikely to be viable, as migrating existing software configurations is costly, and may not even be possible due to restrictions imposed by institution technology providers. Even when the challenges of obtaining source code are overcome, compiling for a different platform may not always be straightforward. Models can depend on operating system specific libraries, preventing compilation unless these library dependencies are removed from the source code. A compatibility layer can allow applications designed for one platform to run on another, for example Wine, which aims to allow Microsoft Windows applications to run on Unix-based operating systems. While not on the same scale as software for scripting GUI interaction, the usage of Wine may involve some configuration, and special care must be taken when supplying Windows file paths on a Unix file system. Although compatibility layer support for individual model applications cannot always be guaranteed, especially where they are from small communities and experience limited usage, utilising a compatibility layer is typically the most desirable and cost-effective solution to communicate with platform dependant models.

3.4.3 Implementing communication

We are faced with several options when implementing communication between model and Web service interface. Typically, a process class on a Web service will implement specific code to communicate with a model. However, if models are executable in a similar manner, for example through the same interpreter, it may be possible to abstract common functionality. Such functionality can be realised as a ‘connector’ — a portion of code, commonly packaged as a library, which allows communication with a model or modelling environment in a completely programmatic manner. The characteristics of the model or modelling environment will determine how generic the connector can be, and the number of models it can be used for. This section explores communication implementation options for different model types, including compiled language models, interpreted language models written in MATLAB and R, and those using a standard modelling interface.

Compiled language models

Communicating with compiled language models can involve significantly more effort than those executable through a standard environment, making it even more beneficial to abstract model communication. A connector for a compiled language model will typically include several classes to represent the various inputs and outputs of a model, and another set of classes to handle communi-

cation with the model. Classes to handle communication are responsible for supplying the model with the necessary inputs, running the executable, and converting outputs to appropriate objects. The mechanisms for data representation and model communication can be diverse, making generic compiled language model connectors practically impossible. However, once a connector is built for a model, this can be used universally, in or out of the Model Web context. Further discussion on communication with compiled language models can be found in Chapter 5.

MATLAB

MATLAB is a language and numerical computing environment for data analysis, algorithm development, and model creation. Functionality can be extended with the installation of toolboxes, which provide additional features, including tools for statistical modelling and analysis. The wide array of available toolboxes increase the appeal of MATLAB for model developers across a broad range of domains.

MATLAB provides access to a Java Runtime Environment (JRE), allowing Java code to be run within the environment, from which any function or variable available in the MATLAB session is accessible. This flexibility can be exploited to run a TCP/IP server from within a MATLAB session, only requiring environment initialisation to be performed once, independently of how many times a model or function is evaluated. However, the interaction with such servers is typically very basic, and can only handle raw MATLAB commands. As such, to communicate with a model written in MATLAB, a Web service interface process must convert all data to MATLAB formatted strings, evaluate the model function, and parse formatted output strings to objects. These steps can be complicated for those unfamiliar with MATLAB and time consuming, especially in cases where a developer has to implement processes for several models.

The `matlabcontrol` library⁸ allows calls to MATLAB from Java, without the use of TCP/IP. The library does not require the MATLAB environment to be initialised before calls are made, and allows all MATLAB interaction to be performed by Java code, in contrast to the basic TCP/IP server approach which requires the server to be started from within MATLAB. Communication from the library to MATLAB is performed using RMI, which can interact with MATLAB on both local and remote machines, the latter requiring additional security setting configuration. While communication is simpler than the TCP/IP approach, as the client deals with objects rather than MATLAB formatted strings, interaction can still be complicated. Retrieving variables from a MATLAB ses-

⁸<https://code.google.com/p/matlabcontrol/>

sion requires the type to be known to make the appropriate conversion within the Java code, and comprehensive support is missing for some data types, such as cells and structures. Additionally, the library is only available for Java developers, and the use of RMI for communication prevents the development of similar libraries in other languages.

To solve these various issues, the matlab-connector Java library was developed, consisting of a server and client component. The server component of the library aims to provide simple, language and platform independent function evaluation, using MATLAB either locally or on a remote machine. The client component aims to facilitate communication with a local or remote MATLAB server, typically to support the automated evaluation of MATLAB models from a Web service interface.

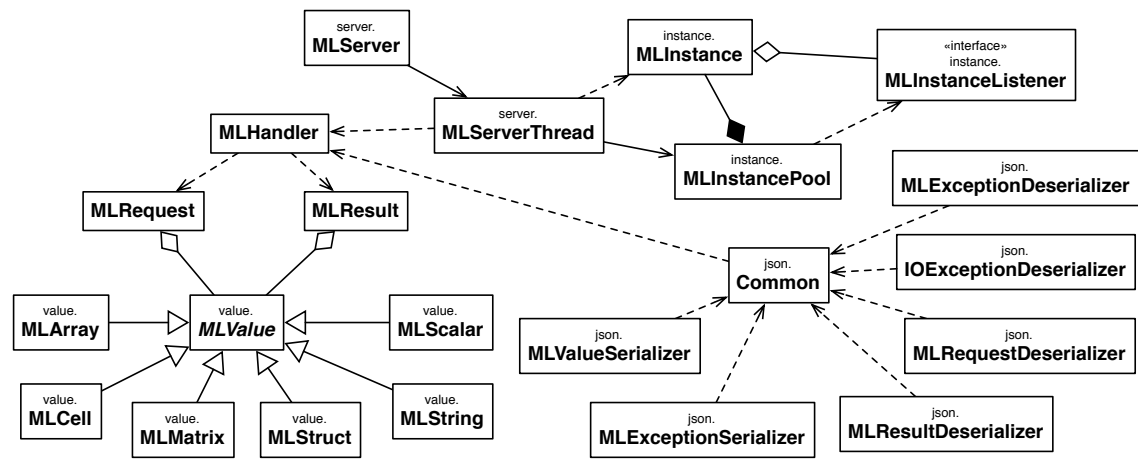


Figure 3.6: A diagram of the main classes comprising the matlab-connector library.

Figure 3.6 shows the main classes comprising the matlab-connector library, including those for representing various types of MATLAB value. In the case of scalar, array, and matrix types, these classes are simply wrappers for the equivalent Java primitives, with support for conversion to MATLAB formatted strings. As cell and structure types do not have direct equivalents in Java, alternative data structures are used for these classes. Implementing concrete classes for value types simplifies client usage, there is no type ambiguity when handling input parameters and outputs. Request, response, and exception classes support communication between client and server, and as with the value classes, can be serialised for network transport.

JSON was selected for data transfer between client due to extensive support in various languages, and the ability to directly encode arrays. However, it was subsequently discovered that JSON may not be suitable for some MATLAB functions, as not a number (NaN) and infinity cannot be represented, and must instead be replaced by null values. Listing 3.8 shows an example JSON request sent to the server component, for a function called ‘MorrisDesign’ expected to re-

```
{ "function": "MorrisDesign",
  "resultCount": 3,
  "parameters": [
    5.0,
    2.0,
    10.0,
    1.0,
    [ [0.0,1.0], [0.0,1.0] ]
  ] }
```

Listing 3.8: A JSON encoded matlab-connector request.

```
{ "results": [
  [ [0.444,0.555], [0.444,0.444], [0.555,0.444], [0.111,0.777], [0.222,0.777],
    [0.222,0.666], [0.444,0.0], [0.444,0.111], [0.333,0.111], [0.222,0.111],
    [0.222,0.0], [0.333,0.0], [0.0,0.111], [0.0,0.0], [0.111,0.0] ],
  [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,1.0], [0.0,1.0],
    [1.0,0.0], [0.0,1.0], [1.0,0.0], [0.0,1.0], [1.0,0.0] ],
  0.111
] }
```

Listing 3.9: The JSON encoded matlab-connector response to Listing 3.8.

turn three output parameters, and called with five input parameters. The first four input parameters are scalar values, and the fifth is a two-dimensional matrix. Listing 3.9 shows the related response from the server, where the first two outputs are two-dimensional matrices, and the third is a scalar value.

The server component was originally developed as an extension to the basic TCP/IP server approach. A server instance is started through a MATLAB function, which is responsible for creating a socket, listening for a connection, and handling the request when a connection is made. After the requested function is evaluated the type of each output parameter is inspected, and the appropriate object is instantiated to represent the MATLAB value. Finally, the server sends the JSON serialised response to the client. As the standard MATLAB distribution is limited to a single computational thread, the server cannot handle concurrent requests, therefore blocking any request made when the server is busy.

Later developments to the library overcame many existing limitations. The server component was rewritten in Java, utilising the `matlabcontrol` library to communicate with a MATLAB instance using RMI. While each instance is still limited to a single computational thread, `matlabcontrol` allows us to start multiple instances. Upon receiving a request, the server component searches for a free instance to process the request. If no free instances are found, the request is queued. In previous methods, unless a function or model has been explicitly programmed for multi-core

processors, only a single core will be used, thus wasting valuable processing power. Utilising multiple instances can ensure that additional cores are used for simultaneous requests. The use of `matlabcontrol` also removed the need to start server instances from within MATLAB, instead allowing all interaction with MATLAB to be performed from the Java server code. Although the server component has been written in Java, its use does not require any knowledge of the language, as the server can be started from a compiled Java Archive (JAR).

The Java client automatically handles object serialisation and JSON deserialisation when communicating with a server instance, allowing users to execute MATLAB functions dealing only with Java objects. Using the client, a process class on a Web service interface can call a MATLAB model with ease, being responsible only for conversion from standard Web formats to MATLAB value objects, constructing and sending a function evaluation request, and translating output parameters into Web formats. Although the client is restricted to the Java language, the use of compact JSON formatted requests and responses enables straightforward local or remote MATLAB function execution from any language.

R

R is a language and environment for statistical computing. R is open source software, and provides a plethora of statistical techniques for modelling, time-series analysis, classification, and clustering. The software is highly extensible, and a variety of packages have been developed to provide additional functionality. These characteristics drive its popularity amongst modellers.

As R is an interpreted language, the environment acting as an interpreter must be started and initialised for each model evaluation. This step, and the overhead it introduces, can be bypassed for sequential model runs through use of a third-party library, `Rserve`. `Rserve` is a TCP/IP server which allows programs written in various languages to use the functionality of R, without the need to initialise R or import R libraries directly. The use of TCP/IP also enables remote model communication, removing the requirement for a model to be hosted on the same machine as the Web service interface. Allowing unrestricted remote access to the R environment can be extremely dangerous, as certain functions can modify the file system. However, an `Rserve` instance enforces a security policy, where a valid user name and password is required before a client can connect. The TCP/IP communication protocol implemented by `Rserve` is fully documented, allowing clients to be implemented on any platform. However, client-side libraries are already available for many languages, including Java, Python, C++, and Ruby, enabling a user to bypass handling low-level

TCP/IP communication, and instead deal with high-level language objects and methods.

Models written in R will typically be functions, requiring the process on the Web service interface to communicate with Rserve to assign inputs to variables, evaluate the function, and retrieve return values. However, if the function handles file-based inputs and outputs, the process on the Web service interface must issue additional commands to R. Once the inputs have been assigned to variables, the process class must instruct R to write the variables to file, in the required format. The model function can then be evaluated, supplying the location of the written files as a parameter. When the function has finished evaluation, the process can instruct R to read the output files on disk, and assign the data to variables, which are subsequently accessible through Rserve and consequently the Web service process. Without this additional code, the process class would require remote file access on the machine where the model is to be run, increasing deployment overhead and potentially introduce security concerns.

While the `matlab-connector` library enables programmatic function evaluation in MATLAB, and Rserve allows interaction with the R environment, the Web service process class is still required to perform a significant amount of work to expose a model, including specifying the identifiers and data types for inputs and outputs, converting between standardised Web formats and model compatible data, and control over the execution phase. As the types of parameters in MATLAB or R functions cannot be determined by their source code definition alone, automatically creating Web service process specifications and performing data conversion is impossible. This would be possible however, if function definitions were annotated to provide the required information to a Web service interface. These annotations must specify, at a minimum, the types of the input and output parameters, but may additionally provide metadata, most preferably using the tags defined in Jones et al. (2012).

The use of annotations to simplify the exposure of models implemented in R has been explored within the UncertWeb project (Nüst and Pebesma, 2012). This exploratory work included developing the ability to upload annotated R scripts through a Web browser, which would then be deployed as WPS processes. Listing 3.10 shows an example of an R script annotated for deployment on a WPS. Once uploaded, the deployed process, ‘numberAdded’ will have inputs ‘a’ and ‘b’ accepting double values, and return a single output ‘result’. This mechanism is currently exclusive to the 52°North WPS implementation, in a backend feature known as WPS4R⁹, but could be integrated into the processing service framework.

⁹<https://wiki.52north.org/bin/view/Geostatistics/WPS4R>

```
# wps.des: numberAdder, title = The number adder,  
# abstract = Optimised function for adding two numbers together;  
  
# wps.in: a, double;  
# wps.in: b, double;  
  
# wps.out: result, double;  
  
result = a + b
```

Listing 3.10: A simple R script annotated for deployment on a WPS.

Although the addition of annotations requires changes to the file containing the model code, it does not require any changes to the model code itself. This approach is likely to be much more appealing for model owners wishing to expose their models on the Web, as they will not be required to develop a process class in a language they may not be familiar with. There are still some issues however with data mapping however, as it may be required to automatically convert complex Web data formats containing several properties, such as O&M, to more primitive MATLAB and R data types.

Standard modelling interfaces

Standardised modelling interfaces can drastically reduce implementation effort required to expose models on the Web. When model behaviour is guaranteed, a single, generic, Web service process class can be used to communicate with all models implementing the interface. A strategy for OpenMI integration with the processing service framework is detailed by Gupta et al. (2012), which is based on mapping methods in the `AbstractProcess` class to OpenMI model equivalents. For example, when the `getInputIdentifiers` method is called from the Web service, `getInputExchangeItem` is called on the underlying OpenMI model, which returns an object containing an identifier for the input.

The studied approach demonstrates potential for simplifying the process of exposing models on the Web, only requiring models to implement a standard modelling interface. If models are automatically wrapped and exposed by a Web service process, a developer requires no previous Web service experience to expose their OpenMI model on the Web. To simplify the process further, it would be possible to extend the service to enable model upload, automatically deploying the model as a Web service once uploaded. However, the generic mechanism requires further development, as it currently only supports OpenMI version 1, and remains untested on complex models, such as those implementing timestepping features.

3.5 Summary

A number of fundamental usability issues with the WPS specification provided motivation to develop an alternative: the processing service framework. The Java-based framework can allow process developers to focus on the implementation of the process, rather than standards for communication and data representation on the Web. In addition to a SOAP/WSDL interfaces, the framework provides a JSON-based interface, supporting the rapid development of Web applications. Fully adopting WSDL and XML enables processes to be developed on the framework to be consumed in a variety of software and tools, without requiring additional modifications to the service or process classes.

The restrictions on implementation time for the framework resulted in several important missing features, all of which are included in the WPS specification. Asynchronous process execution is extremely important for long-running models and geospatial processes, which will be added in future versions. Support is also missing for complex metadata, which may consist of XML with several properties to describe an input or output. This is not possible using the current tag-based system, and may require an additional operation to query metadata from a process, or the use of WSDL-S annotations.

The tests performed only evaluate the usability of the framework in a single contrived use case. While this is sufficient for testing usability in tools and software, it only considers two types of data, GML and double-precision numbers. A more detailed group of tests would involve exposing several diverse models, and working with a variety of data types and usage patterns, something infeasible on the time-scales for this work. This level of diversity may help to explain why the WPS is as generic as it is. Chapter 5 achieves this to some degree, through detailing the use of the framework to expose several models, used to compose a workflow for a real scenario.

In developing an alternative to the WPS, we understand that we are currently lacking in the wider-scale interoperability which is critical to the success of the Model Web. However, the framework was developed as a proof of concept, to show that it is possible to integrate widely used standards with geospatial processes. This has been achieved, as the use of the framework in our use case has demonstrated the additional usage opportunities gained.

Due to the slow and verbose standardisation process in the OGC, a wait of several years can be expected before new technologies are added to a specification. As it is relatively simple to implement such support, developers may do so, albeit in a non-standardised manner. This can create interoperability issues if users create clients based on non-standardised practices, as developers

could implement the support in a variety of ways.

As a result of the work described in this chapter, including the implementation of a simple Web-based geospatial processing scenario, we have identified the advantages, disadvantages, and potential future development bottlenecks for SOAP/WSDL, JSON-based, and WPS Web service technologies. An overview of these characteristics is shown in Table 3.4.

Exposing models on the Web is a huge challenge, one often underestimated when describing the implementation of the Model Web. The sheer diversity of models, modelling interfaces, and platforms results in significant effort required to connect a model to a Web service interface. The difficulty of such implementation will depend on who is responsible for developing the Web service interface. For many model owners, the effort required will need to be met with significant gains from becoming part of the Model Web, something we hope is provided by the tools in Chapter 4.

This chapter introduced several implementation options to provide communication between Web service interfaces and models, aiming to further reduce development time for process developers. These options include a library to enable MATLAB function execution either on a local, or remote machine — thus not requiring computationally expensive models to be evaluated on the same server hosting the Web service. A similar mechanism for R, Rserve was also discussed. The ability to automatically deploy MATLAB and R models on the Web using script annotations, thus removing the need to develop model-specific communication, was briefly explored, with WPS4R demonstrating how this is possible for R and WPS integration. Communication can also be simplified if models comply with a standardised interface, such as OpenMI. The standardised behaviour of such models enables a single, generic, Web service process class to be developed, immediately compatible with all compliant models. While this is extremely beneficial for existing OpenMI models, or where new models are developed, legacy compiled models still remain a challenge, as they still require model specific communication, either in the Web service or model interfaces.

Developing libraries to connect to models and modelling environments can help to minimise the effort required to connect a model to a Web service interface, while the processing service framework simplifies model exposure on the Web. Even with these libraries and the framework, significant effort still be required to expose a model on the Web. The majority of this effort could be removed by automatic interface generation, where annotations within a model are read, and inputs and outputs are mapped to appropriate data formats. However, such interface generation mechanisms are still in their infancy, not suitable for compiled language models, and require either

	Advantages	Disadvantages	Future bottlenecks
SOAP/WSDL	<ul style="list-style-type: none"> • Broad client and server support. • Compatible with workflow tools and software. • Strongly described messages. 	<ul style="list-style-type: none"> • Not standardised for geospatial processing services. • Requires extensions for semantic metadata support. 	<ul style="list-style-type: none"> • Community shift towards simpler technologies such as JSON.
JSON	<ul style="list-style-type: none"> • Maps naturally to programming language data types. • Format of choice for browser-based applications. • Less verbose than XML. 	<ul style="list-style-type: none"> • No standardised service description technology. • Schema languages rarely used. 	<ul style="list-style-type: none"> • Limited adoption by the OGC. • Potentially fragmented data formats.
OGC WPS	<ul style="list-style-type: none"> • Accepted standard in the geospatial community. • Several fully-featured server implementations. • Support for metadata and asynchronous processing. 	<ul style="list-style-type: none"> • Limited client support. • Difficult to integrate with workflow tools and software. • Weakly described messages. 	<ul style="list-style-type: none"> • Slow standardisation process leading to a failure to embrace new technologies. • Restricted progression through maintaining backwards compatibility.

Table 3.4: An overview of Web service technologies.

the model owner or user to provide the annotations.

4

Emulation in the Model Web

CONTENTS

4.1	Introduction to emulators	106
4.2	A tool for emulator building	109
4.2.1	Backend API	109
4.2.2	Frontend Web application	116
4.3	Using an emulator	123
4.3.1	Emulator representation	124
4.3.2	Emulators as Web services	125
4.4	Extensions for SA and validation	128
4.4.1	A tool for SA in the Model Web	128
4.4.2	Supporting additional validation scenarios	130
4.5	Limitations	132
4.6	Summary	134

Chapter 3 concluded that the implementation effort required to develop Model Web components must be met with sufficient user benefits over existing architectures and model usage scenarios. This chapter introduces one potential benefit for model owners and users — generic tools for emulation in the Model Web.

Section 4.1 presents the motivation to create tools for emulation in the Model Web. The design, development and use of a tool for building emulators is detailed in Section 4.2. The tool consists of two components: a Web backend providing standard access to methods supporting emulation (Section 4.2.1), and a Web frontend to guide a user through the emulator building process (Section 4.2.2).

Section 4.3 introduces mechanisms for emulator use, including the inclusion of emulators in Model Web workflows. Section 4.3.1 details a portable format to support the representation and transfer of an emulator. A service to deploy emulators on the Web is implemented in Section 4.3.2, enabling emulators to communicate with other Model Web components, and be integrated with existing workflows.

Reusable functionality provided by the emulator building tool enabled two extensions to be developed in Section 4.4. The first extension allows SA methods to be performed on Web-accessible models (Section 4.4.1), and the second exposes probabilistic validation methods in a user-friendly manner (Section 4.4.2).

Finally, Section 4.5 discusses limitations of both the tools developed in this chapter and underlying emulation techniques.

4.1 Introduction to emulators

Chapter 3 discussed the practical challenges of building Model Web components, and while connectors can simplify communication between the model and Web service interface, exposing existing models on the Web still requires significant implementation effort. For model owners and users, this effort has to be worthwhile. In addition to the benefits of model and data sharing, and complex workflow composition, the Model Web creates potential for generic tools (Nativi et al., 2012). As models and data sources should communicate in a standard manner, a tool can guarantee support for a set of Web service interfaces and standard data formats, rather than individual models and custom data types.

When performing uncertainty analysis using Monte Carlo based methods, a large sample of outputs is needed (Oakley and O’Hagan, 2002), thus requiring a high number of model evaluations.

This is infeasible, even in cases where a model may only take a second to evaluate. One solution would be to perform multiple evaluations of the model in parallel, therefore reducing the physical time needed to perform the runs. As models are computationally expensive, each evaluation will typically be performed on a separate machine, thus requiring access to a large number of resources, or a grid computing platform. A model must be modified to support parallel evaluation on a distributed computing environment — a requirement which may be particularly unwelcome for developers who are additionally responsible for Web-enabling a model.

Emulation, a method introduced in Section 2.2.3, is an alternative solution that can provide substantial computational efficiency gains (Conti and O’Hagan, 2010), allowing Monte Carlo analyses to be performed in seconds, rather than hours, or even days. An emulator is platform independent and does not require a distributed, or even powerful computing environment to run. As a relatively new method, available literature focuses on the underlying statistical concepts and challenges related to emulation, including emulation of non-deterministic functions (Boukouvalas et al., 2009) and correlation between multiple outputs (Rougier, 2008) from simulators. The accessibility of such methods is therefore limited, as although users need a mathematical background, they may not be familiar with many of the underlying statistics they are required to understand the implementation of emulator methods (O’Hagan, 2006).

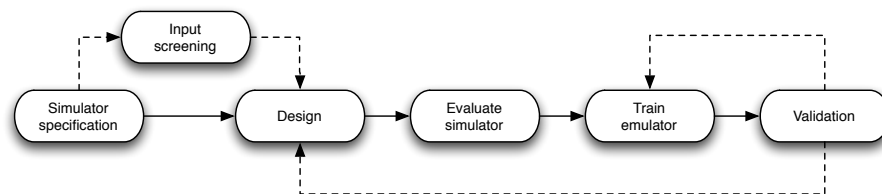


Figure 4.1: The stages making up the emulator building and validation process.

The emulator building and validation process consists of several stages, illustrated in Figure 4.1. Initially, a user must elicit ranges of the model inputs, which may be informed by model documentation, expert elicitation, or knowledge of the problem domain. In cases where an input does not vary, a fixed value can be set. Fixed values are only for simulator evaluation purposes, and are not considered during the emulator training stage.

After specifying these ranges, a user can optionally perform input screening. As described in Chapter 2, screening is a form of qualitative SA which identifies model inputs that have negligible effect on the output. These are known as inactive inputs and can be set to a single, fixed value, thus helping to reduce emulator complexity.

Using the ranges defined in the simulator specification stage, a training design is created. A

design contains a user-specified number of samples, also known as points, for each input. These points should cover the space defined by the range of each input, which can be achieved by generating the design using a method such as Latin hypercube sampling (LHS). In the next stage, the generated design is evaluated against the simulator, which may be implemented as a Web service process, resulting in a set of input points with corresponding simulator outputs.

This set of input and output points form the main input to the emulator training stage, which also includes other parameter adjustments, such as mean and covariance functions, whether to perform normalisation for numerical stability, and the addition of nugget variance to approximate simulator noise. The training stage consists of optimising emulator parameters based on the training design and corresponding simulator run. Once trained, the emulator predictions can be compared to the simulator outputs. A validation design is generated in the same way as the training design, and should ideally not contain any of the same points used to train the emulator. This design must be evaluated against both the emulator and simulator, after which probabilistic validation methods can be applied to diagnose the emulator. If required, emulator parameters can be adjusted, or the size of the design increased. Finally, when validated, the emulator can be used as a surrogate for the simulator, and can potentially be used with UA and SA, or in existing clients and workflows.

These steps involve communicating a large amount of data between the model and the emulation code. For example, after building an appropriate experimental design, the simulator must be evaluated over that design, and the resulting model outputs are passed to the emulator training stage. There are parameters to set at each stage, with many of the parameters strongly dependant on the behaviour of the simulator we wish to emulate. Each stage of the emulation process produces a result, such as a generated training design, optimised length scales, or validation metrics. Ideally, a user will wish to visualise these results, as they will typically contain a large number of values, and will be impossible to analyse without appropriate visualisation.

There are a small number of tools available to support the emulation process. One such tool, GEM-SA¹, is a GUI-based application for performing UA and SA using emulation techniques. Although the software hides complex underlying calculations from the user, the tool still has a high usage barrier. As the tool is decoupled from specific models and modelling interfaces, the user is required to perform several data format transformations to export a training design and import the resulting model evaluation results. The tool also lacks in visualisation options, something which

¹<http://www.ctcd.group.shef.ac.uk/gem.html>

is critical to analyse the emulator training results.

Implementations of emulator methods are also available for MATLAB and R². The MATLAB emulation library allows emulators to be built and validated, with several metric plots and visualisations provided through the process. As with GEM-SA, the library is decoupled from specific models and modelling interfaces, requiring data transformation steps when handling non-MATLAB models. The functions provided by the library are much less accessible to those unfamiliar with the environment than a full GUI-based application, but are more customisable given expertise.

4.2 A tool for emulator building

Motivated by the limited usability of existing tools, the efficiency gains emulator methods can provide, and the potential for generic tools in the Model Web, a tool for building and validating emulators was developed. The tool is aimed at making emulators more accessible and exploits Model Web interoperability to enable model compatibility, independently of model implementation or platform. The tool consists of two components: a backend API to perform all major calculations and data processing, and a frontend to guide a user through each step in the emulator building process. To demonstrate the functionality of this tool, a simple model was exposed using the processing service framework (Chapter 3). While the model, known as ‘SimpleSimulator’, has input and output names to mimic crop yield estimation (Tables 4.1 and 4.2), the underlying calculations are arbitrary and do not result in realistic yield estimation output. The use of the tool with a realistic yield estimation model is described in Chapter 5.

4.2.1 Backend API

The backend API individually exposes all calculations and data processing operations involved in the emulator building process on the Web, allowing the functionality to be used independently of the frontend application. Providing an API creates opportunities for emulator methods to be integrated with existing software, or for new functionality to be developed based on operations exposed by the API. The API can be called in numerical software such as MATLAB and R, enabling a user to interact directly with the data in their preferred environment, should they wish.

²<http://cran.r-project.org/web/packages/emulator/>

Identifier	Description
Rainfall	Daily rainfall average (mm/day)
MeanAirTemperature	Mean air temperature (°C)
MaxAirTemperation	Maximum air temperature (°C)
MinAirTemperature	Minimum air temperature (°C)
MeanSoilN	Mean soil nitrogen content (g/kg)
MaxSoilN	Mean soil phosphorus content (g/kg)
MinSoilN	Mean soil iron content (g/kg)
MeanSoilC	Mean soil carbon content (g/kg)
MaxSoilC	Maximum soil carbon content (g/kg)
MinSoilC	Minimum soil carbon content (g/kg)
MeanSoilTemperature	Mean soil temperation (°C)
MaxSoilTemperation	Maximum soil temperation (°C)
MinSoilTemperation	Minimum soil temperation (°C)
MeanSoilWettingRate	Mean daily soil wetting rate (mm/day)
MaxSoilWettingRate	Maximum daily soil wetting rate (mm/day)
MinSoilWettingRate	Minimum daily soil wetting rate (mm/day)
SoilPorosity	Soil porosity (l/m ² s)
SurfaceAlbedo	Surface albedo
WheatSystolicPressure	Wheat systolic pressure (Pa)
Evapotranspiration	Daily evopotranspiration (mm/day)

Table 4.1: A list of inputs for the ‘SimpleSimulator’ model.

Identifier	Description
WheatGrowthRate	Rate of growth of wheat (mm/month)
FinalYield	Final yield (kg/m ²)
LargestWheatKernelSize	Size of largest wheat kernel (mm)

Table 4.2: A list of outputs for the ‘SimpleSimulator’ model.

Architecture

Web-accessibility of the backend API is enabled by Java Servlet technology. The backend is not currently exposed using the processing service framework, as the two were developed in parallel. However, the API is a good example of an application, external to geospatial models, that could be based on the framework. JSON was selected for data interchange, primarily with the aim to support the Web frontend. Upon receiving a request through HTTP, the backend parses incoming JSON to an appropriate object representing the request. Operations provided by the API are listed in Table 4.3. Functionality classes are then called, passing parameters extracted from the request. Once the computational work has been performed, a suitable response object is built, serialised, and returned to the client as JSON. The components to expose the API on the Web are decoupled from the classes used to perform data processing, enabling the API to be used as a Java library for integration with other applications.

MATLAB and R are utilised by the API for the majority of computational work. These environments were selected as libraries are available for performing the required functionality, and while this dependency does limit usage of the API as a standalone library, porting the required functionality from MATLAB and R to Java would be extremely time consuming. Interacting with code in these environments from the Java backend was made considerably easier by the connectors detailed in Chapter 3, especially with the matlab-connector library, which enabled the API to communicate with MATLAB using Java objects, rather than dealing directly with MATLAB data types.

As a demonstration of the service implementation independence in API requests, simulators exposed using both the processing service framework and WPS are supported by the tool. While support for these services is currently based on the fixed message pattern they prescribe, it would be possible to implement some generic parsing for WSDL, thus additionally supporting all SOAP services described by WSDL documents, providing they have compatible data types. The architecture has been developed with extensibility in mind and does not couple any API functionality to service interfaces, allowing additional service interfaces to be supported when required.

When making requests to the API, a user does not need to be concerned with how the model is exposed on the Web. The API adopts a generic set of classes to represent service descriptions, simulator and emulator input/output data, and communicate with models exposed on the Web, enabling a client to interact with the API independently of service interfaces. This hides the complexity of communicating with Web services from the user, allowing them to focus on using

Operation	Purpose
GetProcessIdentifiers	Retrieve a list of supported process identifiers from a Web service.
GetProcessDescription	Retrieve a description of a process on a Web service.
Screening	Perform input screening on a Web service process.
Design	Create a design for a set of inputs.
EvaluateProcess	Evaluate a Web service process against a design.
TrainEmulator	Train an emulator using a design and process evaluation result.
EvaluateEmulator	Evaluate an emulator against a design.
Validation	Compare simulator and emulator evaluation results.

Table 4.3: A list of operations provided by the emulation API.

the emulation methods while the API has responsibility of understanding how to communicate with the services themselves.

API operations

The API provides two operations, `GetProcessIdentifiers` and `GetProcessDescription`, to retrieve a generic description of a service. A `GetProcessIdentifiers` request consists of only a service URL, and returns a list of process identifiers available on the service, excluding those which are not compatible with the tool. A `GetProcessDescription` request consists of a service URL and a process identifier, and returns a description of the process with the given identifier. Listing 4.1 shows an example `GetProcessDescription` request and response. The response contains a list of input and output descriptions, each containing an identifier and the type of the variable. The type does not refer to the data encoding, but underlying characteristics of the variable. If provided by the service, a text summary of the parameter, units of measure, and the minimum and maximum permissible values are also included in the description. These descriptions are created by classes extending `AbstractProcessDescriptionParser`, of which two currently exist — one for the processing service framework and one for WPS.

Before building a design to train the emulator, the `Screening` operation can be performed. The API exposes the Morris screening method (Morris, 1991) and involves creating an optimal experimental design developed specifically to identify inactive inputs with a small number of points, thus reducing the number of model evaluations required. An experimental design is represented by the `Design` class, which maps arrays of sampled points to input identifiers. This class was extended by `MorrisDesign`, a specialised design which adds additional fields for Morris parameters, and a static method to create a Morris design from a set of inputs. After a design is created, the process is evaluated.

```
// Request
{ "type": "GetProcessDescriptionRequest",
  "serviceURL": "http://uncertws.aston.ac.uk:8080/ps/service",
  "processIdentifier": "SimpleSimulator" }

// Response
// Input and output arrays have been truncated
{ "type": "GetProcessDescriptionResponse",
  "processDescription": {
    "identifier": "SimpleSimulator",
    "inputs": [
      { "identifier": "Rainfall",
        "description": {
          "detail": "Daily rainfall average.",
          "dataType": "Numeric",
          "uom": "mm/day",
          "minOccurs": 1, "maxOccurs": 1 },
        "range": { "min": 0, "max": 10 } }
    ],
    "outputs": [
      { "identifier": "WheatGrowthRate",
        "description": {
          "detail": "Rate of growth of wheat.",
          "dataType": "Numeric",
          "uom": "mm/month",
          "minOccurs": 1, "maxOccurs": 1 } }
    ]
  } }
```

Listing 4.1: A GetProcessDescription request and response for the SimpleSimulator process.

As requests are not required to include any reference to how the processes are exposed on the Web, such as whether they use a processing service framework or WPS interface, the API is responsible for building multiple requests to send to a service, and orchestrate the execution of these requests. This is handled by classes extending `AbstractProcessEvaluator`, which, as with the process description parser, exist for both the processing service framework and WPS. A process evaluator is supplied a service URL, process identifier, a set of input and output specifications, and a design. A single method, `evaluate`, is responsible for evaluating the process against the design, thus performing all communication with the process. A design only contains points for variable inputs, but as process may require other parameters to execute, any fixed values are retrieved from the set of input specifications. An *input specification* can be for either a variable or fixed valued input. In the case of the latter, the specification must contain a single value which is required to evaluate the process. The results of process evaluation are returned as a `ProcessEvaluationResult` object, which shares much in common with a `Design` object, but instead maps arrays of values to output identifiers. The size of the arrays are equal to the size of the design the process was evaluated against. Listing 4.2 shows an `EvaluateProcess` request and response for the `SimpleSimulator` process. In this example, input specifications for ‘MeanAirTemperature’ and ‘MaxAirTemperation’ do not contain a range or a value as they are included in the design. However, as the ‘WheatSystolicPressure’ input is not included in the design, its specification contains a single fixed value.

Once the process has been evaluated, the effects of each input are calculated and returned. Morris design generation and analysis is performed with MATLAB using a set of functions created by Alexis Boukouvalas at Aston University. Metrics computed by the Morris method, including the elementary effects of each input variable, are returned to the caller.

The `Design` operation builds a LHS design given a set of inputs. These inputs must be specified with an identifier, minimum and maximum value, and the request must indicate the required sample size. The sampling is performed with MATLAB, using the `rand` and `randperm` functions. An object representing the design is built by the API and then returned to the operation caller. The object contains a point array for each input, and the points can be individually retrieved using an input identifier. This operation can be performed independently of process or Web service. A design forms the key input for the `EvaluateProcess` operation, which exposes the previously discussed process evaluation mechanisms.

Emulator training requires a design, the result of evaluating the simulator against that design,

```

// Request
// Design map and inputs arrays have been truncated
{ "type": "EvaluateProcessRequest",
  "serviceURL": "http://uncertws.aston.ac.uk:8080/ps/service",
  "processIdentifier": "SimpleSimulator",
  "design": {
    "size": 3,
    "map": [
      { "inputIdentifier": "MeanAirTemperature",
        "points": [11.836510794413178, 5.128624683963153, 5.838045438697931] },
      { "inputIdentifier": "MaxAirTemperation",
        "points": [16.92571098266804, 8.447859462492211, 11.355581262363312] }
    ]
  },
  "inputs": [
    { "identifier": "MeanAirTemperature" },
    { "identifier": "MaxAirTemperation" },
    { "identifier": "WheatSystolicPressure" , "value": 0.6},
  ],
  "outputs": [
    { "identifier": "WheatGrowthRate" },
    { "identifier": "FinalYield"},
    { "identifier": "LargestWheatKernelSize"}
  ] }

// Response
{ "type": "EvaluateProcessResponse",
  "evaluationResult": [
    { "outputIdentifier": "WheatGrowthRate",
      "results": [23.850938102033368, 46.92437235104972, 19.998060896347752] },
    { "outputIdentifier": "FinalYield",
      "results": [24.69702091467573, 14.939022069330886, 17.898773061796668] },
    { "outputIdentifier": "LargestWheatKernelSize",
      "results": [78.53541446153649, 20.903271748600794, 29.793323640956455] }
  ] }

```

Listing 4.2: An EvaluateProcess request and response for the SimpleSimulator process.

and a set of parameters. A mean and covariance function must also be specified, in addition to a length scale multiplier. Emulator training is performed with MATLAB, using the `gpmlab` library created by Remi Barillec at Aston University, after which optimised parameters are returned to the caller.

By treating all inputs and outputs on the same scale, normalisation can help stabilise emulator numerics, thus producing better emulators. Functionality for normalisation is provided by the API, through the use of additional parameters in the request message. An existing `Design` or `ProcessEvaluationResult` can be normalised, with or without a given mean and standard deviation, and can then be subsequently unnormalised. Providing a mean and standard deviation is essential to ensure that future designs evaluated with an emulator are normalised using the same parameters as the original training design.

Validation is critical to ensure the emulator accurately represents behaviour of a simulator over the specified range of inputs. A validation request will contain optimised emulator parameters, a validation design, and a service URL and process identifier for the Web-accessible simulator. The design is evaluated against both emulator and simulator. If the simulator has already been evaluated against the validation design, a request can substitute the service URL and process identifier for the process evaluation results. Validation metrics are computed using MATLAB code developed by Dan Cornford at Aston University.

4.2.2 Frontend Web application

Architecture

The frontend was originally developed as Grails application, a Web framework based on the Groovy programming language. As Groovy runs on the Java Virtual Machine (JVM), this would allow us to integrate the backend Java API directly with the frontend. However, due to its maturity and extensive third-party library support, development was restarted using the Ruby on Rails Web framework. Ruby on Rails uses the Model-View-Controller (MVC) architecture pattern and adheres to REST principles when accessing resources. For example, POST is used to create new resources and PUT is used to update existing resources. The wealth of third-party libraries, known as ‘gems’, enabled rapid implementation of layouts, form generation, API communication and background processes.

The frontend has been designed in a manner that does not tightly couple classes to the API. Each stage of the emulation process is represented as a model class, and these classes refer to other

classes representing individual results. Each class representing a stage is required to implement two methods: one for generating a JSON request to send to the API and another to handle the response received. If changes are made to the API, only these two methods are affected. Instances of these classes exist in two states: the first when they are initially created, where they only contain parameters to send to the API, and the second after the API response has been handled, where they contain the parameters and results.

API integration

Any class using API functionality is referred to as being a ‘remotable’ class. A `Remotable` module, shown in Listing 4.3, was developed to allow such classes to inherit common characteristics, including fields to store the current status of remote communication with the API. A `RemotableController` is also defined, which acts as a generic controller for any remotable class. This controller exploits reflection features of the Ruby programming language to make dynamic references to fields and methods named after the current remotable object. For example, views typically rely on the controller to set instance variables containing the objects they display. To assist readability in the view, these variables will generally share the same name as the class. However, as the `RemotableController` handles classes with several names, we must set these instance variables using a special method, as shown in Listing 4.4, which allows the variable name to be specified as a string, equal to the singular form of our class name.

```
module Remote
  module Remotable
    extend ActiveSupport::Concern

    included do
      field :proc_start_time, type: DateTime
      field :proc_end_time, type: DateTime
      field :proc_status, type: String
      field :proc_message, type: String
      validates_inclusion_of :proc_status, in: ["queued", "in_progress", "error", "success"], allow_blank: true
    end

    def queued?
      self.proc_status == "queued"
    end

    # Remaining status methods removed for listing
  end
end
```

Listing 4.3: The `Remotable` module included by classes using API functionality.

```

def edit
  # find object and set instance variable
  set_instance_object(instance_constantized.find(params[:id]))

  # translates to:
  # @design = Design.find(params[:id])
  # @input_screening = InputScreening.find(params[:id])
end

private

def set_instance_object(object)
  instance_variable_set("@#{instance_singular}", object)
end

def instance_singular
  controller_name.classify.underscore.singularize
end

def instance_constantized
  controller_name.classify.constantize
end

```

Listing 4.4: Using reflection to create dynamically named variables in the `RemoteController`.

Certain stages of the emulation process can take minutes, or even hours to perform on the API. While the number of simulator runs is significantly lower with emulation techniques than traditional UA and SA, hundreds, or even thousands of evaluations are still needed to build a good training design. It is crucial the Web application user is not required to leave their browser open, or have any other resources occupied until the API work is complete. Therefore, the frontend Web application must execute and manage long-running jobs asynchronously.

`Delayed::Job`³ is a database-backed asynchronous queue system for Ruby. With `Delayed::Job`, any fragment of code can be added to a queue for asynchronous execution. Jobs are executed by one of a user-specified number of daemons, which run as separate processes on a system. Integration with `Delayed::Job` is implemented by the frontend in a generic manner. When creating a remotable resource, `RemotableController` generates the necessary JSON request, initialises and then enqueues a custom job for the remotable object. This job responds to `Delayed::Job` events, such as job success or fail, and updates the status fields in the object accordingly. When the job has completed successfully, the `handle` method in the object is called, which parses the response from the API and stores the results in the object.

While some processes may take several minutes to complete, others will take seconds. In these cases, a user is highly likely to keep their browser open and the frontend must therefore

³https://github.com/collectiveidea/delayed_job

display appropriate messages to keep the user informed of progress. To prevent users from having to manually refresh, a polling mechanism was implemented, where the client-side JavaScript code will repeatedly check the current status of the job. This utilises features of the Ruby on Rails framework, where instead of requesting the remotable resource as HTML, we request JavaScript. This JavaScript either tells the browser to display an ‘in progress’ message and request the same resource again in two and a half seconds, or to render the result. If the job was unsuccessful, the result will be an error message and an option to try the request again is provided.

Building an emulator

The Web application groups emulation stages into a project which represents the training and validation of a single output emulator. A project belongs to a user, and access control only allows the project to be viewed or modified by its owner. An emulation project is initially created by specifying a service URL, after which the frontend will issue a `GetProcessIdentifiers` request to the API. The user can then select a model to emulate from the resulting list of identifiers of compatible processes. Upon selection of a model, the frontend sends a `GetProcessDescription` request to retrieve the list of inputs and outputs. These are used to create a simulator specification — a model of the input and output descriptions. The remainder of this section will describe each stage of the frontend, illustrated by training and validating an emulator for the ‘SimpleSimulator’ model.

Figure 4.2: Updating input descriptions in the simulator specification.

The emulation process starts by editing the simulator specification, as shown in Figure 4.2,

where the user is required to select, for each input, whether the value can be given as a range or fixed value. A range will be used to build designs for input screening, emulator training and validation, while fixed values will only be used to evaluate the simulator. If provided by the process description, additional input metadata, such as units of measure, is presented to the user.

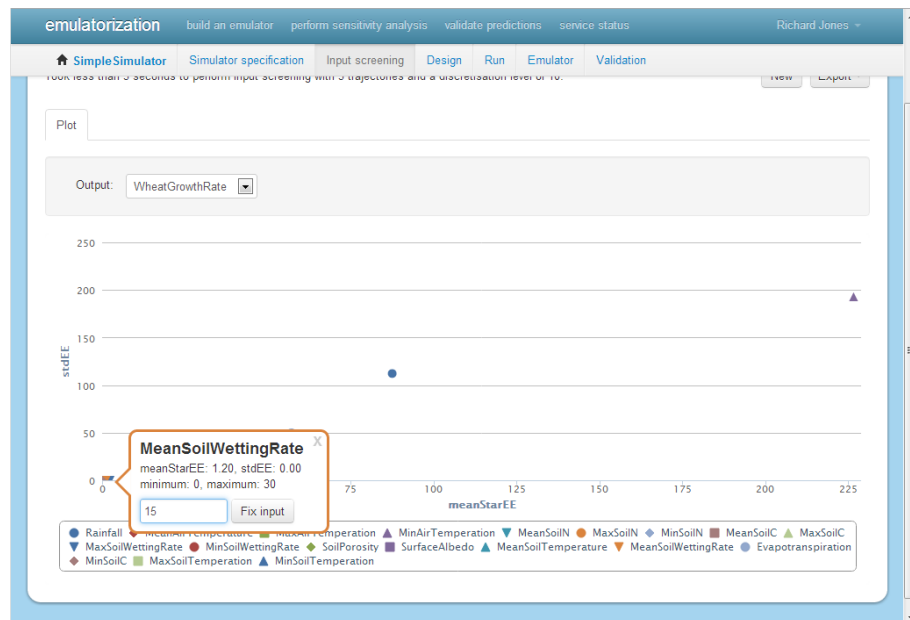


Figure 4.3: Fixing an inactive input from the screening results view.

Once the simulator specification is complete, a user can perform screening to help identify inactive inputs. Inactive inputs can be excluded from the training design, thus reducing emulator complexity. Elementary effects values calculated by the API are added to an interactive plot, which allows adjustment to the level of zoom and visibility of inputs. After the user has identified inactive inputs, they can set an input to have a fixed value by clicking the point, as seen in Figure 4.3. Once set to a fixed value, the input will only be used to evaluate the simulator, and will not form part of the training design.

When specifying the size of the design to create for emulator training, a sensible default value is automatically set by the Web application, equal to the number of inputs multiplied by 15. To enable the emulator to be trained and validated multiple times without having to re-evaluate the simulator, the generated design points will be split between training and validation stages. The multiplication factor of 15 is to cover 10 times the number of inputs for training points, and 5 times the number of inputs for validation points. For example, if we have a simulator with 20 inputs, the default design size is 300, with 200 of those points used for training the emulator and 100 for validation. After a design has been generated by the API, histograms of the resulting points for each input are displayed, as shown in Figure 4.4. These histograms can help to ensure points

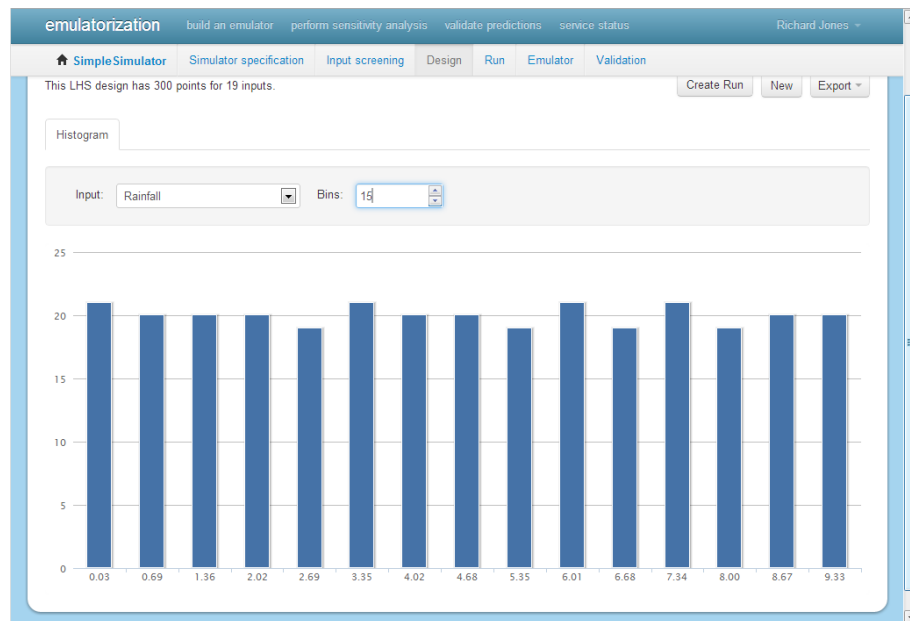


Figure 4.4: Histogram of the design points for input with identifier ‘Rainfall’.

in the generated design cover input space evenly. If so, the bars for each histogram bin will be the same height. A user may wish to generate a larger design if the space is insufficiently covered.

Once a satisfactory design has been generated, the simulator can be evaluated. Depending on the size of the design, this will typically take longer than other stages, proving an effective demonstration of the benefits provided by the asynchronous job mechanism. Upon completion, histograms of the resulting values for each output are available. Using these histograms, a user can check whether the specified input ranges are producing sensible simulator outputs.

Emulator Train an emulator.

Output:

Training size: (The remainder can be used for validation.)

Normalisation: ☒ Enabled ☐ Disabled

Mean function: ☒ Zero ☐ Constant ☐ Linear ☐ Quadratic

Covariance function: ☒ Matern ☐ Squared exponential

Length scale multiplier: (This value is multiplied by the standard deviation of each input.)

Nugget variance: ☒ Enable

Figure 4.5: Setting emulator training parameters.

As the frontend and API currently only support single output emulation, a user must select which output they wish to emulate. Figure 4.5 shows emulator training parameter setting, with the ‘FinalYield’ output selected to emulate. Other parameters can be set, including the size of the design used for training, whether to apply normalisation, a choice of mean and covariance functions, a length scale multiplier, and whether to include nugget variance for numerical stability or to model simulator randomness. Once training has completed, optimised parameters are returned to the user. It is possible to export the emulator at this stage, using the format discussed in Section 4.3.1, but validation should be performed first to ensure the trained emulator accurately represents the simulator.

Validation is performed by evaluating the emulator against the remainder of the design after training, and comparing the results with the corresponding simulator evaluation results. As both the training and validation designs are evaluated before the emulator training stage, validation does not require simulator evaluation, thus drastically reducing the time to perform validation on the API. A wide array of validation metrics are provided, most of which are displayed using interactive plots.

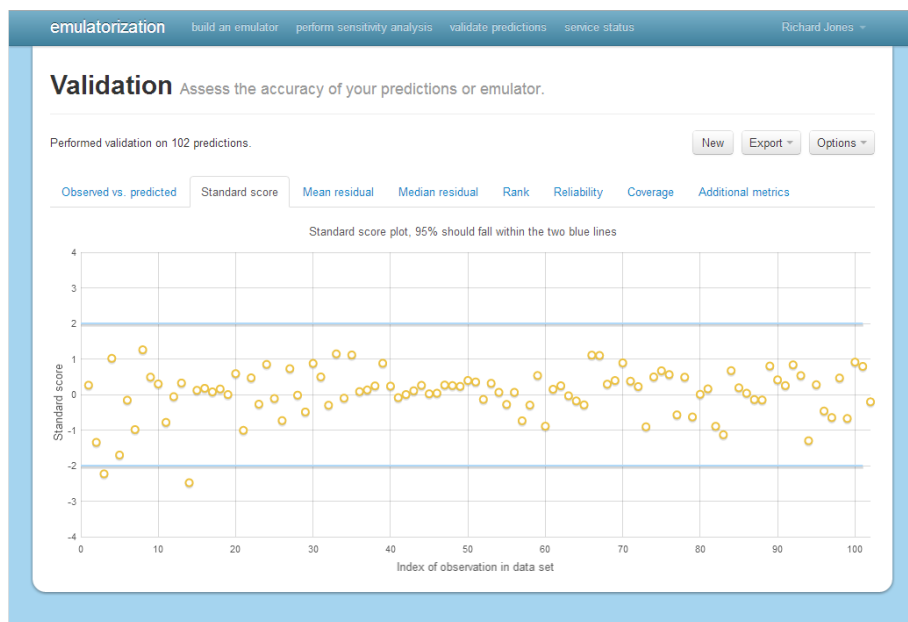


Figure 4.6: Standard score plot for the ‘SimpleSimulator’ example emulator.

The standard score plot can help determine whether the emulator is over or under confident. Figure 4.6 shows a standard score plot generated for the ‘SimpleSimulator’ example, and has the majority of scores between the two blue lines, indicating an under-confident emulator. Typically, a good emulator has 95% of scores between these lines. Figure 4.7 shows a mean residual histogram and QQ plot, for the purpose of checking the distribution of emulator errors. The visualisations

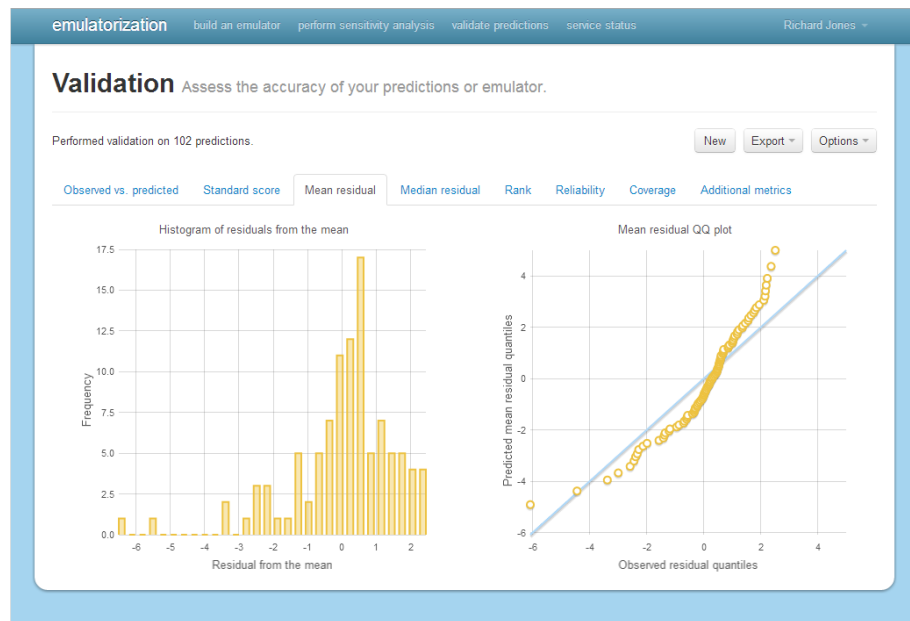


Figure 4.7: Histogram and QQ plot of mean residuals for the ‘SimpleSimulator’ example emulator.

provided at this stage are for diagnostic purposes — they help to inform the user if the emulator is not a good fit to the model, but also the reasons why it is not a good fit. If the results are unsatisfactory, a user can return to the training stage, adjust parameters, and train the emulator again. If the results are still unsatisfactory, a user can try increasing the size of the design, or fix certain inputs to provide emulator stability.

While the frontend provides a variety of visualisations to assist the user, advanced users may wish to perform additional data analysis. Results from the majority of the stages, including designs, runs, training, and validation can be exported in formats suitable for subsequent importing into MATLAB and R, and also a JSON object for use in other programming languages.

4.3 Using an emulator

Once trained and validated, an emulator can be used in a variety of applications. These applications include, but are not limited to, UA, SA and workflow evaluation. To support the usage of such applications in the Model Web, an emulator should be transferable in a standard manner, and appropriate mechanisms are required to enable the use of emulators as surrogates for the Web-accessible simulators they were trained to represent.

4.3.1 Emulator representation

A trained emulator consists of a design, associated process evaluation results, and a set of optimised parameters, allowing the emulator to be stored in a compact format for transfer over the Web. These common emulator characteristics are a significant advantage over traditional models, which are very heterogeneous in nature. Additionally, there are overheads associated with transferring traditional models, which may consist of several large files. As API requests and responses are JSON encoded, JSON was also chosen as the primary export format for an emulator built with the frontend Web application.

```
// Input and output arrays, design, evaluation results and length
// scales truncated
{ "inputs": [
  { "identifier": "MeanSoilTemperature",
    "range": { "min": 3.0, "max": 12.0 },
    "description": {
      "dataType": "Numeric", "encodingType": "double",
      "detail": "Mean soil temperation.",
      "uom": "degreesC" } }
],
"outputs": [
  { "identifier": "WheatGrowthRate",
    "description": {
      "dataType": "Numeric",
      "encodingType": "double",
      "detail": "Rate of growth of wheat.",
      "uom": "mm/month" } }
],
"design": {
  "size": 198,
  "map": [
    { "inputIdentifier": "MeanSoilTemperature",
      "points": [0.12286961958590968, 1.014858285916736, -0.49348594750649194],
      "mean": 7.358562058933247,
      "stdDev": 2.5751766558823657 }
  ]
},
"evaluationResult": [
  { "outputIdentifier": "WheatGrowthRate",
    "results": [-0.9072129009609949, 1.8136357480538703, -0.6815210086083802],
    "mean": 53.232593648632616,
    "stdDev": 44.643230023985915 }
],
"meanFunction": "linear",
"covarianceFunction": "matern",
"lengthScales": [10.79333579584265, 10.899840684359317, 5.017686549129081] }
```

Listing 4.5: JSON representation of the trained emulator for ‘SimpleSimulator’.

A JSON emulator representation has been designed for use by both the API and frontend, and

can potentially be adopted by external tools and software. Listing 4.5 shows the emulator trained for ‘SimpleSimulator’ encoded as a JSON object. To enable an emulator to be used in the same manner as the simulator, a JSON encoded emulator contains a description of inputs and outputs, including input ranges, fixed values, data types, and any additional metadata originally provided by the simulator. The design and process evaluation results used for emulator training follow the same format adopted for API requests and responses, which are essentially arrays of objects, with each object containing a design or results for a named input or output. Mean and covariance function names are lower-case, underscore delimited strings, for example ‘squared_exponential’ and ‘matern’. Optimised length scales for each input are represented as an array, and the optional nugget variance is a single double value.

While the chosen structure can successfully represent an emulator in a compact and transferable manner, it will require further changes to enable widespread adoption of the encoding in tools and applications. As they are merely string values, mean and covariance function names must be fully defined in a dictionary to ensure consistency between implementations. Due to time constraints, a dictionary has not been developed for this purpose. The representation of a trained emulator could be enhanced with additional lineage information, such as details of the simulator the emulator was trained to represent, who trained it and when. This information may be essential for the Model Web, where emulators could potentially be discovered automatically.

4.3.2 Emulators as Web services

When emulators are represented in a compact and transferable format, they can potentially be evaluated in a variety of programming languages and environments. However, this is not a straightforward process, and requires advanced knowledge of the emulator structure and Gaussian processes. Libraries and tools can alleviate some of this burden, but such libraries are rare, and do not currently provide direct support for emulators encoded in the JSON format developed in Section 4.3.1. Ideally, an emulator should be exposed on the Web, enabling emulator evaluation and workflow inclusion in the same manner as the Web-enabled simulator used for training. An emulator compatible processing service was developed to expose emulators on the Web. This can be considered to be similar to transactional extensions that allow BPEL workflows to be exposed as WPS processes (Schäffer and Foerster, 2008), but instead of BPEL workflows, emulators can be uploaded.

A transactional extension for emulator deployment

The service is based on the processing service framework described in Chapter 3. A transactional process extension was developed to allow emulators to be uploaded and exposed as processes. Emulators are uploaded through the `UploadEmulator` process, which has inputs for the desired process identifier and the emulator itself. Within the `run` method, a custom `ProcessRepository` is instructed to add an emulator-based process. The custom `ProcessRepository` extends the default repository with transactional features. When instructed to add an emulator-based process, the repository adds the process and stores the emulator in a database. MongoDB⁴ was chosen to store transactional processes, as its document-based, schema-free nature enabled transactional features to be implemented without having to define a database structure. Upon initialisation, the repository adds processes for emulators stored in the database, in addition to processes specified in the configuration file.

A generic process class, `RunEmulator`, is instantiated for each uploaded emulator and added to the list of processes held by the repository. This differs from traditional processes, as it has an argument-based constructor. As multiple emulators cannot share the ‘RunEmulator’ process identifier, the `RunEmulator` constructor is passed an identifier for the process and the emulator itself. A class representing an emulator is present in the API, and was used in the data description for the emulator input. Appropriate encoding classes were developed for the service, utilising existing classes in the API. As the backend API can already evaluate an emulator, this functionality was not duplicated. Instead of adding an extra communication layer between the emulator upload service and the API, the API was imported as a library, allowing the upload service to evaluate an emulator directly.

After being uploaded to the service, an emulator is accessible as a standard SOAP/WSDL and JSON-based Web service, and can thus be used as a direct surrogate for a simulator in the Model Web. This demonstrates the extensibility provided by the framework. Through implementing a small number of generic process classes and extending the existing process repository, emulators could be uploaded in a dynamic manner, and subsequently available for execution, as with any other Web service process.

An emulator will generally provide faster evaluation of a single run than a simulator. However, emulation provides huge increases in speed when evaluating a design consisting of several points, for example, when performing UA. Using a simulator, this will typically be performed by repeat-

⁴<http://www.mongodb.org/>

edly calling the Web service to evaluate the model. The statistical nature of an emulator allows a multi-point design to be evaluated in a single pass. This is supported by the service interface, where instead of accepting a single value for each input, the process requester can specify an array of values for each input. The resulting output is therefore no longer a single distribution, but an array containing the results for each set of input values. Comparisons of emulator and simulator performance in a real case study are detailed in Chapter 5.

By default, evaluating an emulator exposed on this service returns a distribution, or set of distributions. It is also possible to have the service generate a given number of samples from these distributions, through use of the `RunEmulatorSamples` process. This process is accessed by concatenating the identifier of the exposed emulator with ‘Samples’. This choice of output may be more desirable when considering uncertainty-enabled workflows, as only services which are uncertainty-enabled will be able to accept a distribution as an input, requiring those which are not to accept a single realisation drawn from the emulator output.

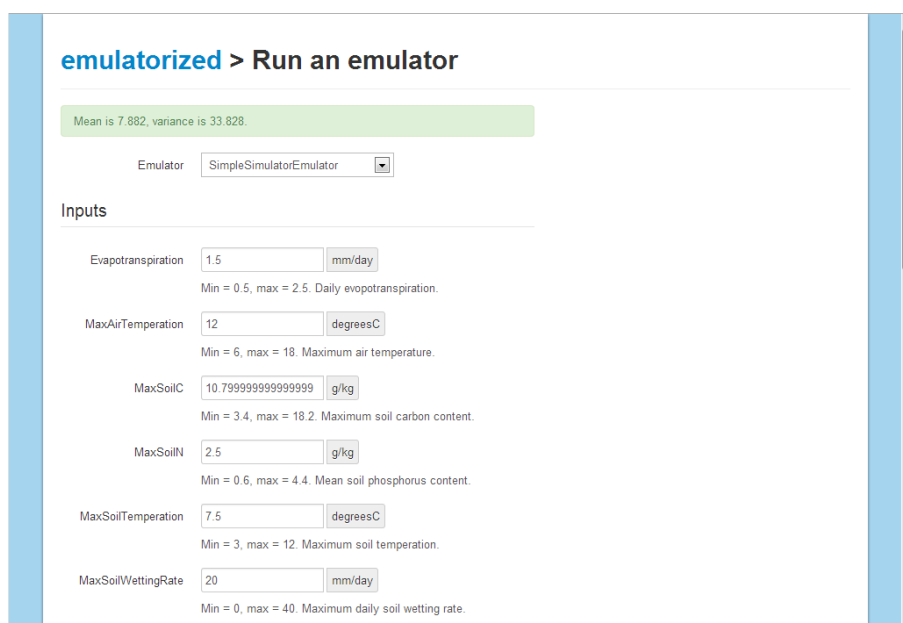


Figure 4.8: Running the emulator for ‘SimpleSimulator’ using the upload service client.

To demonstrate the emulator upload service, a simple Web-based client was developed as part of the extension to the processing service framework. The client provided the facility to upload a new emulator, or to run one already uploaded to the service, as shown in Figure 4.8. After training an emulator using the frontend, a user can export the trained emulator as a JSON object, then copy the object to the emulator input on the upload emulator page. Due to the standalone nature, and lack of integration with any other Model Web components, the run emulator page on the simple client is likely to be less of use than the upload page. However, it can still serve as a mechanism

Operation	Purpose
Sensitivity	Perform SA with a Web service process or emulator.
Validation	Compare predictions to observations, with or without an emulator.

Table 4.4: Additional and modified API operations developed for SA and validation.

for demonstrating the emulator. Fields for input values are dynamically generated from the service description retrieved from the service, and if minimum and maximum value metadata is present, the default value of these fields is set to the middle of the specified range.

4.4 Extensions for SA and validation

The extensible nature of the frontend and API allowed new functionality to be developed for SA and validation. Table 4.4 summarises the API operations added and modified to support the new functionality.

4.4.1 A tool for SA in the Model Web

The use of SA, a mechanism for attributing uncertainty in a model output to uncertainty in model inputs, can be enhanced by emulation methods. As with UA, SA typically involves thousands of repeated model executions, thus benefiting greatly from the faster evaluation times offered by emulation techniques. As with the emulator building process, current mechanisms for performing SA are limited, and none support Web-enabled models. To solve these issues, the existing API and frontend for emulation was extended to support SA using both Web-enabled simulators and emulators, with the aim to provide a generic tool suitable for use throughout the Model Web.

The Web application groups SA stages into a project. Upon creation of a project, a user is prompted to select whether they wish to perform SA using either a Web-enabled simulator or emulator. If a simulator is selected, the user must provide a simulator specification, in the same manner required for emulator building. The ranges contained in a simulator specification are used by the underlying SA methods to create a design for evaluation. If an emulator is selected, the frontend provides a list of validated emulators for the user to select from. Once the simulator specification is complete, or an emulator has been selected, a user can proceed to SA parameter selection, as shown in Figure 4.9. These parameters include the size of the design to generate during analysis, the SA method to use, and any method specific parameters, such as the number of bootstrap replicates and target confidence level used by the Sobol method.

The screenshot shows a web interface for 'emulorization'. At the top, there are navigation links: 'emulorization', 'build an emulator', 'perform sensitivity analysis', 'validate predictions', and 'service status'. The user 'Richard Jones' is logged in. The main section is titled 'Analysis' with the subtitle 'Quantify the contributions of different uncertainty sources.' Below this, there are input fields for 'Design size' (set to 200) and 'Analysis Method' (with 'Sobol' and 'Fast' buttons, 'Sobol' is selected). Under 'Sobol parameters', there are fields for 'Number of bootstrap replicates' (set to 100) and 'Confidence level for bootstrap confidence intervals' (set to 0.95). A 'Create Analysis' button is at the bottom of the form.

Figure 4.9: Selecting the Sobol SA method and setting associated parameters.

On the API, the `Sensitivity` operation provides a standard way to perform Sobol and FAST SA methods. As there is no universal formula for measuring sensitivity, different measures have different uses and applications (Saltelli et al., 2000), thus providing the option of two methods can increase applicability of the tool. A `Sensitivity` request must contain either a simulator URL, process identifier, and input and output specifications, or simply an emulator, encoded according to the format defined in Section 4.3.1. The simulator or emulator details are accompanied by the size of the design to generate, the SA method name, and method specific parameters. Once a request is received, the API is responsible for design generation, simulator or emulator evaluation, and subsequent analysis. When generating a design, the API utilises the R sensitivity package⁵. Initially, the API passes the design size and input ranges to R, which generates a design using the requested method. This design is retrieved by the API, evaluated against the simulator or emulator, the results of which are sent to R for analysis. Analysis results, which typically include main and total effects for each input, are retrieved by the API, and returned in the response to the user.

An analysis is a remotable model on the frontend, and therefore interacts with the API asynchronously. Once the analysis is complete, the results are plotted by the frontend, with different visualisations for Sobol and FAST methods. Figure 4.10 shows the resulting plot of main and total effects calculated by the Sobol method, using an emulator created through the process detailed in Section 4.2.2. The plot indicates high output uncertainty contributions from inputs ‘Evapo-transpiration’, ‘MinAirTemperation’, and ‘MinSoilTemperation’, but with a significant amount of

⁵<http://cran.r-project.org/web/packages/sensitivity/index.html>

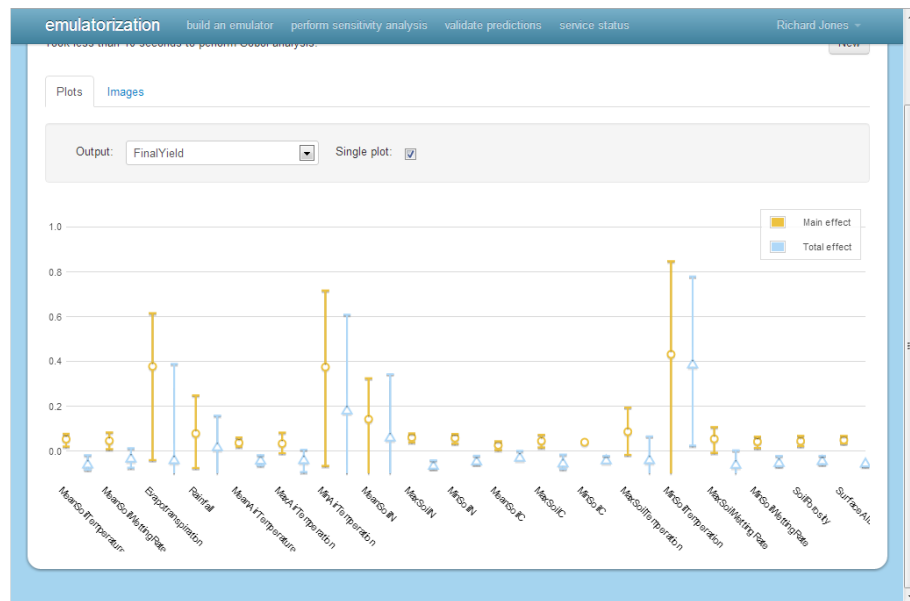


Figure 4.10: Main and total effects calculated by the Sobol method.

uncertainty in the calculated measures, suggesting the design size may be too small for a reliable analysis.

4.4.2 Supporting additional validation scenarios

Although originally developed for assessing the accuracy of an emulator, the probabilistic validation mechanism could provide detailed validation metrics and plots in many other contexts. Traditionally, producing such metrics and plots may be difficult, even for those familiar with probabilistic validation, as the process involves data import, writing functions to calculate metrics, and creating appropriate plots. The challenges associated with performing validation, and the existing mechanism developed for emulators, provided the motivation to extend the API and frontend to support validation in a non-emulation context.

On the API, a `Validation` request was modified to allow different types of observed and predicted values, and remove the previous emulator requirement. While the set of observations must contain scalar values, predicted values can be normal distributions or samples, also often known as ensembles in the weather forecasting domain. As the results from the process evaluation are scalar values, and the results from the emulator evaluation are normal distributions, the underlying validation code remains the same when supporting these additional scenarios. When encoded as JSON in the request, scalar values are passed as an array, normal distributions are passed as an array of objects, each with a mean and variance field, and samples are passed as an array of arrays. These representations were chosen for simplicity and rapid implementation, and a transition to

UncertML, to support existing standards, is a possibility for a future improvement.

No changes were required for the emulator validation stage on the frontend, as a validation request sent to the API can still contain an emulator, design, and process evaluation result. To support additional scenarios on the frontend, a validation section was added alongside existing emulator building and SA sections. As with emulator building and SA, validation is project based, and employs a remotable model for calculating metrics asynchronously using the API. Observed and predicted values are currently importable through a CSV file, but standard interchange formats, such as O&M, could be added in the future. CSV was selected as the tool was developed in cooperation with UncertWeb partners Norwegian Institute for Air Research (NILU), who served as usability testers and required CSV import, for their probabilistic air quality forecast system.

Validation Assess the accuracy of your predictions or emulator.

Import from CSV

Data:

Identifier type: ☐ Auto ☐ Single column ☒ Multiple columns

Variable type: ☐ Scalar ☐ Distribution ☒ Ensemble

First identifier column: First ensemble column:

Last identifier column: Last ensemble column:

datum	hour	station	cmpd	ens01	ens02	ens03	ens04	ens05	ens06	ens07	ens8
20110402	1	hoogvliet	no2	41.824585	46.751228	46.08247	46.895988	52.361328	53.081135	43.372971	46.301
20110402	2	hoogvliet	no2	43.088467	44.695988	47.365059	46.245346	50.37812	50.559521	43.572163	48.61
20110402	3	hoogvliet	no2	45.544319	40.093449	47.007729	41.268768	49.301552	44.808186	47.87849	42.41
20110402	4	hoogvliet	no2	39.230965	39.34491	40.709278	39.036648	48.007236	44.835243	45.765675	40.09

Figure 4.11: Importing predicted sample values from a CSV file.

Users can select a CSV file on their local machine, which is then uploaded to the frontend and subsequently displayed to the user, as shown in Figure 4.11. From the CSV display window, a user can select whether the CSV file contains observed or predicted values, and which columns contain the values to be used for validation. For example, if the CSV file contains samples, a user must select the first and last columns containing samples. The Web application will assume any columns between those selected also contain samples. If the CSV file contains normal distributions, a user must select two columns containing the mean and variance values. Each row is assumed to contain a single value. Matching between the observed and predicted values can either be done automatically, by implicit ordering or by selecting a column, or multiple columns, which uniquely identify each value. Figure 4.11 shows four columns, 'datum', 'hour', 'station', and

'cmpd' selected to uniquely identify a single predicted value, and columns 'ens01' to 'ens50' containing ensembles of each predicted value. The standalone validation result shares the same view as the emulator validation, as both sections store results in the same model, ensuring code is not repeated.

In addition to utilising UncertML to represent normal distributions and ensembles in the API, and supporting O&M in the frontend, the validation mechanism could be extended to add further integration with the Model Web. Currently, validation without an emulator requires the simulator predictions to have already been made. Using instances of `AbstractProcessEvaluator` on the API, it would be possible to evaluate the simulator within the validation request, providing the request contains a service URL, process identifier, and design to evaluate. This would allow simulator validation to be performed in a single step, rather than having to evaluate the model separately.

4.5 Limitations

Implementation time constraints combined with the complexities of data encoding formats and emulation techniques resulted in several limitations to functionality provided by the API and frontend, thus reducing the applicability of the tools developed.

A model may have input parameters or data sources which remain fixed between runs. Although these values can be fixed on the service interface or model, this is not possible in cases where the model is provided by a third-party, or the fixed parameters are specific to a single scenario. Unless these inputs have double values, even though they are fixed, the frontend and API cannot currently support such models. Future improvements could allow such inputs to be set as a fixed value, thus supporting a wider array of models. The API is merely required to copy these fixed values into a request, and the frontend must provide an appropriate form input in the simulator specification, such as a toggle button for a boolean valued input. If the input uses complex data types, the frontend could simply allow the user to specify a data reference URL.

Web service processes using complex data formats such as O&M are currently not supported by the tools. When evaluating a process, the API automatically builds and sends request documents, setting points from the design as input values. For complex data formats containing several properties, this is not as simple as setting a single value. Using O&M as an example, while the API could theoretically generate an O&M Measurement, setting the value of the result property to a design point, the API cannot set meaningful values for other properties, such as the observation

time and feature of interest. In certain cases, these properties may not actually be used by the model, only acting as metadata, but it is impossible for the API to determine instances where this is true. However, it would be possible to allow a user to indicate such cases through the use of a GUI.

While it would be possible to implement this basic level of O&M generation for cases where the user knows the model only requires the result property, other complex data types remain a challenge. A similar approach to O&M could be taken, where design points from the design are used as a value to a single element. However, the mechanism for specifying which element should contain a design point and how the rest of the data is generated requires significant development on both the structure of the API request, and the frontend user interaction.

Underlying emulation techniques are also limited. At the time of development, only continuous values were supported in the MATLAB emulation library. However, recent developments have also added support for discrete valued inputs and outputs (Tulloch, 2013). The ability to emulate discrete values would require changes on both the API and frontend, as a user must be able to specify whether an input is discrete or continuous. A discrete value could be numeric (ordinal), or categorical (nominal), and will therefore require modification to the input specification GUI, which currently only supports continuous variables.

A key limitation of the emulation tools, especially when considering the context, is the lack of support for geospatial models. While there are methods for emulating both spatial and temporal models (Rougier, 2008), these are unsupported by the MATLAB emulation library. The same applies for the emulation of multi-output models, which can currently be performed by creating a separate emulator for each output. This approach is time consuming, and will not be suitable if the model produces highly correlated output errors. Therefore, a true multi-output emulator approach, such as that proposed by Conti and O'Hagan (2010) may be more desirable.

The methods provided by the SA extension, Sobol and FAST, may not be appropriate for all types of model behaviour. Further applications of SA could be supported by providing additional methods. The extension also has some limitations when combined with emulator methods. If an emulator is used for SA, any computed measure will be affected by additional uncertainty introduced by the emulator. The `Sensitivity` operation currently makes no distinction between SA measures calculated using an emulator, or a simulator, and does not therefore take full account of emulator uncertainty and the efficient methods for SA proposed in Oakley and O'Hagan (2004).

When performing validation, a user is limited to performing validation on continuous valued

observations and predictions supplied as CSV files. If the predictions are supplied as distributions, the tool currently assumes a Gaussian distribution, and samples accordingly. Future improvements would make it possible to support both continuous values and sampling from a variety of distributions, while also adding the ability to supply observations and predictions in other formats, such as O&M.

4.6 Summary

The tools developed for emulation, SA and validation represent a significant contribution to the Model Web. Although emulation and SA techniques have been studied for several decades, the usage of such techniques has been limited, restricted to a small number of tools and libraries for numerical computing environments. The API developed exposes a number of operations on the Web to support emulation, SA and validation, which are utilised by the frontend to provide access to such functionality, all through a user-friendly Web application.

Exposing operations on the API provides access to computational functionality, allowing usage outside of an emulation context. The API can be employed for a variety of purposes, including generating a LHS design, evaluating a simulator, or performing validation on a set of predicted ensembles. API request messages are decoupled from specific Web service interfaces, and allow a user to focus on the emulation and SA functionality they wish to perform, rather than communication with the Web service. An extensible architecture was chosen for the API, potentially allowing further interfaces to be supported without requiring changes to request and response message structures.

The frontend groups the steps required for emulation, SA and validation logically in a single project. Parameters for each stage are specified using forms, with sensible defaults automatically generated where appropriate. The frontend delegates computation to the API, which is requested asynchronously so the user is not required to remain in the Web application whilst waiting for long-running computational jobs to complete. Interactive visualisations are provided for the majority of steps, assisting the user with reaching conclusions that may inform parameter specification in a subsequent stage. Export functionality allows for further analysis in MATLAB and R, and any software or programming language capable of reading JSON.

Although the tools are a significant improvement over existing mechanisms for building emulators, performing SA, and validating predictions, they are not accessible for everyone. While the frontend presents a single emulator building stage at a time, and provides a variety of result visu-

alisations, there is currently little to guide a user through the process. A user must know how, for example, to interpret validation results, and when they should set the nugget parameter when training an emulator. However, as a Web application, the frontend could be easily extended to guide a user through the tool by adding detailed descriptions to project stages, stage parameters, and result data and visualisations. Much of this guidance could be based on the information contained in the Managing Uncertainty in Complex Models (MUCM) Toolkit⁶.

Exposing emulators in a uniform way on the Web presents significant new opportunities to broaden their usage. Until now, emulators have only been accessible locally, requiring specific software or numerical platforms to be installed. Including an emulator in a workflow requires special interaction with this software to pass input data to the emulator and retrieve the output. The emulator upload service enables an emulator to be evaluated in the same manner as the simulator, as a surrogate. Once exposed as a SOAP/WSDL and JSON Web service, an emulator can be used in a variety of tools and software, and can be included in a workflow to interact with existing models exposed on the Web.

The fundamental requirement for usage of the developed tools is that models must be exposed on the Web, currently using either the processing service framework or WPS. Chapter 3 discussed the significant challenges and implementation effort associated with exposing existing models on the Web. Sufficient motivation to undertake this implementation effort we hope will be provided by the tools developed. However, it is not expected that these tools will provide enough motivation in all cases, as users may not require the efficiency provided by emulation techniques, need to quantify the effects of different sources of input uncertainty with SA, or have models compatible with existing methods.

Application of the tools is currently limited by a number of restrictions. Some of these are caused by underlying emulation techniques, some due to implementation time constraints, and others due to usability challenges. Further developments could eliminate several of these restrictions. For example, fixed values in a simulator specification can currently only be double values, making simulators using any other data types incompatible with the tools. However, support for a much wider variety of models could be provided by supporting fixed values of any type, in both the API and frontend. Emulation techniques could then be applied using geospatial models, which may take, for example, a raster map as an input. Although the emulator trained using this method could only be used in cases where the raster map input is equal to the one used for training, users

⁶<http://mucm.aston.ac.uk/MUCM/MUCMToolkit/index.php?page=MetaHomePage.html>

can still benefit from emulator efficiency in these cases. Other restrictions, such as those caused by the huge input spaces associated with emulation of geospatial models, and the usability challenges associated with automatically generating complex input data, remain open research questions and are not in the scope of this thesis.

5

Case study

CONTENTS

5.1	Foreword	138
5.2	Introducing the scenario	138
	5.2.1 Models	138
	5.2.2 Workflow	141
5.3	Web architecture development	142
	5.3.1 Exposing models and data	142
	5.3.2 Building the workflow	148
5.4	Emulation of AquaCrop	151
5.5	Summary	155

5.1 Foreword

Chapter 3 detailed technologies and approaches for exposing models on the Web. This chapter focusses on applying these methods to a real case study, introduced in Section 5.2. Section 5.2.1 contains a summary of the models in the case study, and Section 5.2.2 provides an overview of how these models will be arranged into a workflow to produce the desired output.

Section 5.3 describes the Web architecture developed to realise the case study workflow, and Section 5.3.1 details work undertaken to expose the models on the Web. Section 5.3.2 demonstrates how the workflow of models and data sources was composed and subsequently orchestrated.

Chapter 4 introduced Model Web compatible tools for building and deploying emulators. However, these tools have only been used on contrived models, built only for the purpose of testing. Section 5.4 evaluates the tool use in the context of a real example from the case study workflow: a model for calculating crop yield estimates.

5.2 Introducing the scenario

While the processing service framework, mechanisms for communicating with models, and emulation tools have all been tested with a small number of models during development, testing has considered contrived examples, which may not accurately represent models found in the real world. To enable more realistic usability testing, the framework and tools were used to deploy a case study workflow as part of the UncertWeb project.

The FERA case study aims to explore the effect of climate change on crop yields, and gather an impression of the future agricultural landscape (Johnson et al., 2010). Two English regions are considered in this case study — the East Anglian Chalk National Character Area (NCA) in the south east, and the Yorkshire Wolds NCA in the north east, selected by FERA as they are two complementary and well-studied agricultural regions. The workflow will predict current and future crop yields under uncertain climate scenarios, which can help to inform policy making regarding land management practices, farming behaviours and food availability.

5.2.1 Models

There are three main model components in the FERA case study. These models will be orchestrated in a workflow, described in Section 5.2.2, to help discover whether climate change might

have an effect on crop yields in England. The models are:

- a land capability classification model (generating crop transition probabilities);
- a field use simulator (simulating yearly cropping patterns);
- a crop simulation model (producing estimates of crop yield).

The first model, a land capability classifier, has been developed by Jill Johnson at FERA. This model, known as the Land Capability Classification System (LCCS), accepts a set of field texture observations which describe the type of soil in each field of interest, for example ‘clayey’ or ‘loamy’. Field texture observations are accompanied by historical crop observations, which specify the crops grown in each field for a given set of years. LCCS adopts a Markov process approach to generate an uncertain crop transition matrix for each field texture, which contains a set of probabilities — each representing the chance of a field containing one crop type transitioning to another crop type in the following year.

For the second model, a field use simulator, the LandSFACTS model¹ developed by the Macaulay Land Use Research Institute was selected. LandSFACTS uses a combination of stochastic and rule-based processes to create scenarios of crop or land use within fields and regions. A user must specify a set of probabilities representing crop or land use changes, and can also provide a number of spatial and temporal constraints. LandSFACTS will run field or region level simulations for a given number of years, and generate outputs containing the simulated crop or land use for each field or region, for each simulated year.

AquaCrop², was selected for the final yield calculation model. The model, developed by the Food and Agriculture Organization of the United Nations (FAO), simulates crop development based on several parameters, and calculates crop yields based on these development simulations. A user can specify crop characteristics, soil properties, and climate data, including temperature, rainfall and evapotranspiration. AquaCrop requires a different set of characteristic parameters for each crop type. Acquiring such parameters is a time consuming process, and involves consultation with experts. As a result of these challenges, yield estimates in this workflow will only be produced for fields simulated to contain wheat. Crop characteristics for wheat were gathered by FERA, including ranges where parameters were uncertain.

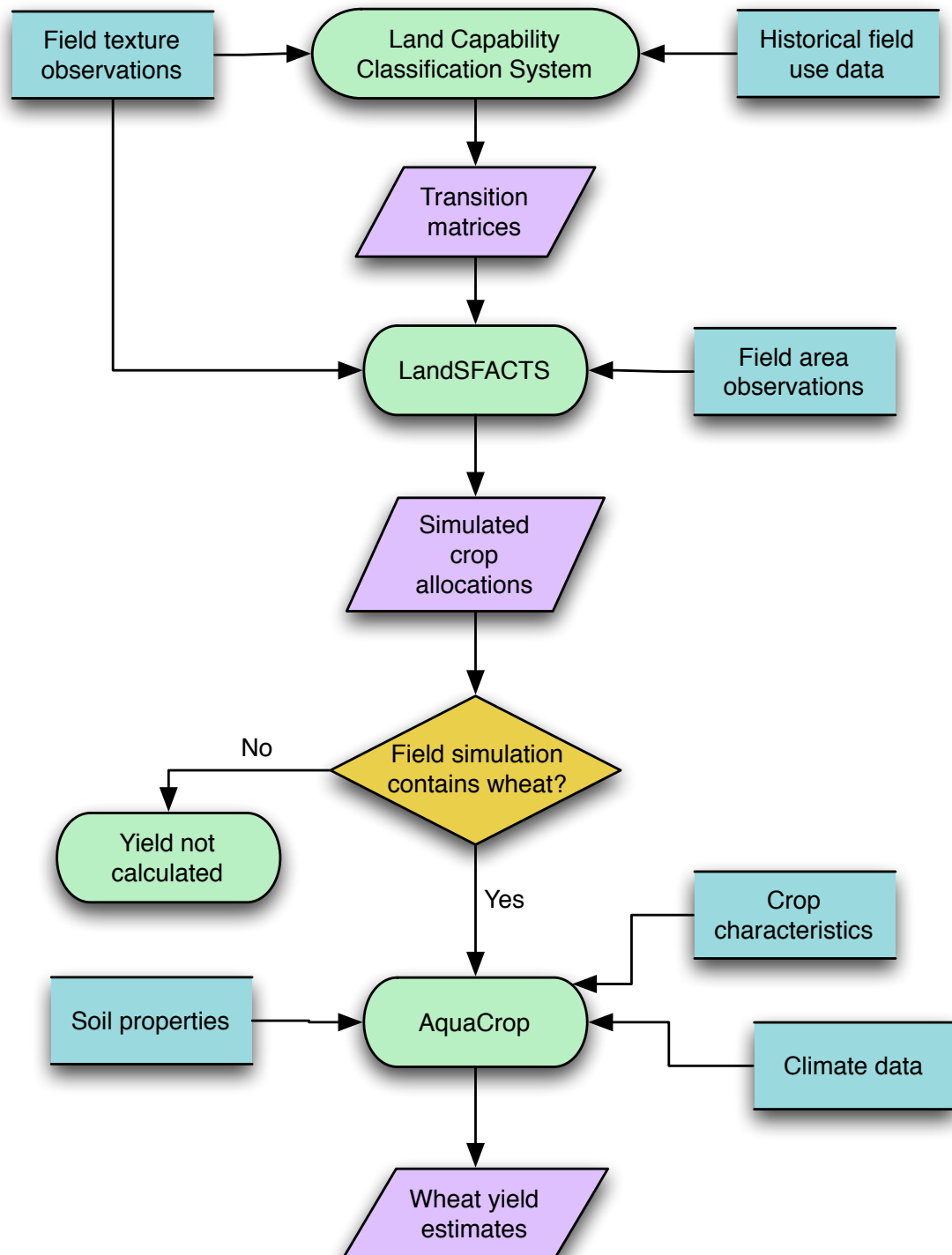


Figure 5.1: A basic workflow composition for the FERA case study.

5.2.2 Workflow

The model components are composed in a workflow, shown in Figure 5.1. Once appropriate data sources are added, the workflow can help to answer the question of whether climate change might effect the production of wheat in England.

Workflow inputs consist of historical field use data, field texture (soils) and area observations, crop characteristics, climate data, and soil properties. The majority of this data must be retrieved by the workflow orchestrator, but some, for example the crop characteristics for wheat, have already been gathered by FERA and thus will remain fixed. Workflow outputs consist of predicted wheat yields for each year of the simulated field use, for each field simulated to contain wheat.

The workflow commences with historical field use data and field texture observations being sent to LCCS. The resulting output, uncertain transition matrices for each texture, form part of the input to the next model, LandSFACTS. However, these transition matrices contain Dirichlet distributions, and LandSFACTS only accepts transition matrices with fixed probabilities. Therefore, Monte Carlo must be performed by sampling from the Dirichlet distributions and executing LandSFACTS for each sample. This also introduces an extra workflow input, the number of samples to draw from the uncertain transition matrices.

LandSFACTS is passed a transition matrix sample, and field texture and area observations. By default, LandSFACTS runs for a period of five years, and runs three simulations. At this stage in the workflow, even though LandSFACTS runs for five years, there is no specific date attached to these years. From the resulting simulated field uses, the workflow orchestrator must select those which contain wheat to send to AquaCrop.

AquaCrop is run for each field, for each year where it has been simulated to contain wheat. Climate and soil properties are selected based on the location of the field. The climate data determines the specific date of the simulation, with the earliest data being assumed to be from the same year as the first in the LandSFACTS simulation. Climate data for both current and future scenarios were generated by the UKCP09 weather simulator. This data is uncertain, and therefore must be analysed accordingly. Uncertainty on the weather data is represented as realisations, allowing us to run AquaCrop using several of these realisations. A combination of the uncertainty introduced by the transition matrices, LandSFACTS, and the climate data will require AquaCrop to be run thousands of times — indicating that the model may be a possible candidate for emulation.

Finally, the workflow orchestrator must combine the output results from multiple simulations.

¹<http://www.macauley.ac.uk/LandSFACTS/>

²<http://www.fao.org/nr/water/aquacrop.html>

For each field, for each year, if the field was simulated to contain wheat, several yield realisations will exist. Executing the workflow with both current and future climate data will allow the differences in wheat yields to be compared, accounting for uncertainty in the generated transition matrices, field use simulator, and climate data. Once the workflow has completed, it would be possible to spatially aggregate the field level results on a regional basis, allowing FERA to analyse and compare overall results for the East Anglian Chalk NCA and Yorkshire Wolds NCA.

5.3 Web architecture development

With no models or data sources currently exposed on the Web, the FERA case study can serve as an example to assess the benefits and challenges associated with exposing models and data on the Web. This section will detail how the FERA models and data sources were integrated with Web services, and how the workflow was defined and subsequently orchestrated.

5.3.1 Exposing models and data

Exposing models on the Web is the most fundamental task in the implementation of the FERA workflow. Without the models being exposed on the Web, the emulation building tools cannot be used, and model execution requires local installation and specific communication mechanisms to pass inputs to, and retrieve outputs from, a model. All of the FERA models will be exposed on a single Web service, which implements the processing service framework detailed in Chapter 3. Developing a Web service using the framework requires the packaged library to be imported, and creation of an empty configuration file and deployment descriptor, the latter of which enables the Web container to call the appropriate service and data servlets when requests are made. The configuration file will eventually contain the fully qualified names of the process classes developed for each model.

As none of the models in the FERA case study can be executed in the same manner, each process class must implement specific functionality to communicate with the model. Section 3.4.3 detailed a typical pattern for such cases, where the process class is responsible for converting the parsed input objects to the format required for the model, executing the model, and converting the model outputs to output objects. This section details individual Web service process implementation strategies for each model in the case study. Implementation strategies may be strongly affected by the platform of the Web service interface and the computational servers, both of which are Linux based in this example.

LCCS

The LCCS model is written in the R programming language. As discussed in Section 3.4.3, we can use Rserve to enable Java code to interact with a remote instance of R. To minimise resource usage on the machine hosting the Web service interfaces, Rserve was set up on our computational machine. While LCCS can be called with an R function, in the same manner as many compiled language models, it deals with file-based inputs and outputs. To enable communication from the Web service process class, this requires either changes to the source code to accept inputs as parameters and return output values from the function, or the process class to generate and parse the input and output files.

Due to a lack of familiarity with the source code and R programming language, the latter option was chosen. The Web service process class assigns values from input historical crop and field texture O&M observations to variables in R, which are then written to file, and the LCCS function can be called. Generated output files are read into R variables, which can subsequently be retrieved using Rserve. From these variables, transition matrix objects are created, with one for each texture class. As there is currently no appropriate format for encoding uncertain transition matrices, a simple format and accompanying schema was developed, utilising UncertML to represent row or cell uncertainties. Listing 5.1 shows an example transition matrix generated by LCCS and encoded using the format.

```
<tm:TransitionMatrix xmlns:tm="http://www.uncertweb.org/transitionmatrix" xmlns:un=
"http://www.uncertml.org/2.0">
  <tm:identifier>loamy</tm:identifier>
  <tm:headings>BA1 BP FH FSA FXLI GR OA1 OAR OLA OSR PO1 SU1 VAS WH</tm:headings>
  <tm:row>
    <un:DirichletDistribution>
      <un:concentration>15.0 8.0 1.0 3.0 2.0 1.0 1.0 3.0 2.0 6.0 2.0 4.0 1.0 1.0</
un:concentration>
    </un:DirichletDistribution>
  </tm:row>
  <tm:row>
    <un:DirichletDistribution>
      <un:concentration>2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 9.0</
un:concentration>
    </un:DirichletDistribution>
  </tm:row>
  <!-- Remaining rows removed from example -->
</tm:TransitionMatrix>
```

Listing 5.1: An uncertain transition matrix generated by the Web-enabled LCCS model.

LandSFACTS

As LandSFACTS is only usually available as a GUI based Windows application, a version unsuitable for automated execution in a Linux environment, source code for the standalone model was requested from the original software authors. While the standalone version could be executed in an automated manner under Windows, the same was not possible in a Linux environment. Upon inspection of the source code, a number of Windows-specific calls are made in the standalone model, such as those to display a dialog box when errors occur during model execution. As LandSFACTS writes all output to disk, including error messages, these calls are unnecessary and were subsequently removed. A standalone model binary could then be compiled under Windows, Mac OS X, and Linux.

Once the model was executable on our chosen platform, the Web service process class could be developed. Developing a process class requires understanding the inputs, outputs and execution parameters for the chosen model. LandSFACTS inputs and outputs are contained in a number of TSV formatted files. There is a single file for each input, and the format uses identifiers to link crop types and fields between each of the files. LandSFACTS is executed with a command-line argument specifying the folder containing inputs, and where the outputs should be generated. The Web service process class would typically be responsible for handling file reading, writing, and model execution. However, to maximise reusability and contribute to the modelling community, a library, `landsfacts-interface`³ was developed to allow Java code to interface with the LandSFACTS model, enabling programmatic execution.

The `landsfacts-interface` library contains several classes to represent LandSFACTS inputs, and a `Project` class grouping instances of these inputs for a given scenario. The model is executable through the `LandsfactsInterface` class, which creates a temporary folder, serialises the inputs in the required TSV format, and executes the model. Once the model has finished execution, the interface parses the TSV formatted outputs to Java objects, which are then returned to the user. This allows LandSFACTS to be executed in an automated manner, without having to handle TSV formatted inputs or outputs and model execution manually. Listing 5.2 shows a simple example of how the library can be used to simulate crop allocations.

With the use of the `landsfacts-interface` library, the Web service process class is only required to create a `Project` object based on the inputs provided, send that project to the interface, and convert the returned outputs to objects compatible with the process. A Web encoding format must

³<https://github.com/itszootime/landsfacts-interface>

```
// Create project
int numSimulations = 3;
Project project = new Project(fieldDescriptions, initialTransitionMatrices,
numSimulations);

// Create interface
LandsfactsInterface landsfactsInterface = new LandsfactsInterface("/path/to/
landsfacts");

// Run
List<CropAllocation> cropAllocations = landsfactsInterface.run(project);

// Get allocation for first field
Allocation fieldAllocation = allocation.getAllocations().get(0);

// Print details
// LandSFACTS runs over a simulation period of 5 years
String[] crops = fieldAllocation.getCrops()
System.out.println("Crop allocations for field " + fieldAllocation.getFieldID());
System.out.println("Year 1 = " + crops[0]); // etc.
```

Listing 5.2: Executing LandSFACTS programmatically with the landsfacts-interface library.

be selected for the inputs and outputs, and the UncertWeb profile of O&M was chosen as the most appropriate for the field texture and area observations, and also the simulated crop allocation output. For the latter, the `UncertaintyObservation` element is utilised, grouping the output from all simulations into a single sample, resulting in a single observation for each field, in each simulation year. The transition matrix encoding developed for the LCCS model was also used for the LandSFACTS initial transition matrix input. A small amount of conversion between the parsed Web service inputs and outputs and objects in the landsfacts-interface library must be performed by the process class.

AquaCrop

Unlike LandSFACTS, the FAO provide two versions of AquaCrop: a standard GUI program, and a ‘plug-in’ version. The plug-in version has no user interface, and is aimed at applications where iterative runs are required. AquaCrop takes inputs as a set of text files, with files for crop characteristics, climate data, soil properties, and an overall project file to link them together. Each of these files are formatted differently. For example, the crop characteristics have a single property value on each line, but a temperature data file contains a header specifying the frequency and first date of the measurements, and subsequent lines containing minimum and maximum temperature measurements separated by a tab. The plugin-in version of AquaCrop produces a single output text file, containing results formatted as a table, which is done in a manner directed towards human

readability rather than machine readability.

As with LandSFACTS, a library, `aquacrop-interface`⁴, was developed to interface with AquaCrop. The `Project` class groups all input data, including crop characteristics, climate data, and soil properties, which is then serialised into the necessary files for AquaCrop to read. Once executed, the interface parses the resulting output file into an instance of the `Output` class, containing, amongst other outputs, the calculated crop yield.

Both the AquaCrop standard GUI program and plug-in version are Windows executables, but are compatible with Wine, a layer allowing applications designed for Windows to run on Linux systems. However, problems are encountered when running the AquaCrop plug-in program on headless machine, such as the computational server where the model is deployed. Although the plug-in program is supposed to have no interface, the program actually creates a single, empty window, upon execution. On headless machines, this will cause the program to terminate unexpectedly. To resolve this issue, the host machine must have an instance of X virtual framebuffer (Xvfb) running. Xvfb enables windowed programs to run, performing all graphical operations in memory, rather than displaying them on a screen.

Further issues are encountered if errors occur during AquaCrop execution. Although the plug-in should have no user interface to enable automation, an error will cause a modal dialog box to be displayed, requiring a user to acknowledge the error message to terminate the program correctly. Without complicated OCR techniques, it is impossible to retrieve the error message displayed by the dialog box, and acknowledging the error message in an automated environment is also a great challenge. A better design for such standalone, interface-free programs, is adopted by LandSFACTS, which writes error messages to file. However, even if source code is available, modifying the original model is likely to be infeasible in this instance. This example serves to illustrate some of implementation challenges faced when exposing models on the Web. In the case of AquaCrop, a reliable mechanism for automating error message acknowledgement could not be found. Instead, the library will forcefully terminate the plug-in program if outputs are not generated after an arbitrary length of time, and throw an exception. Due to the impracticality of reading any error messages, only a standard error message is included in the exception thrown.

As AquaCrop is more computationally expensive than LandSFACTS, running the model on the same machine as the Web service interface may degrade interface performance and response times. Therefore, AquaCrop was executed on the computational server. This requires an extra

⁴<https://github.com/itszootime/aquacrop-interface>

layer of communication, as the Web service process must now interface with a remote AquaCrop installation. The aquacrop-interface library was extended with remote server and client components, using TCP/IP and Java serialisation to transfer `Project` and `Output` instances between server and client. On the server-side, special care was required to handle multiple requests simultaneously. The AquaCrop plug-in program reads input files from the installation directory, and will process any project files it finds. If requests are processed simultaneously, the chances of AquaCrop executing two or more projects is high, resulting in longer execution times. Further issues are encountered when execution of one of those projects fails, as if that project is the first one AquaCrop evaluates, any subsequent projects will fail too. These issues were resolved by copying the directories containing AquaCrop executables for each model run, thus providing a guarantee that AquaCrop will only evaluate a single project per plug-in execution. Executing AquaCrop projects in parallel also enables multiple cores to be utilised, allowing us to take advantage of the sixteen core computational machine.

As AquaCrop is a candidate for emulation, the inputs and outputs of the Web service process must be compatible with the emulation tools developed. AquaCrop has no spatial inputs, meaning the inputs can be single double values, as required for the emulation tools. However, special care was required for climate data inputs, which can be given as daily, ten-daily, or monthly values. Daily or ten-daily values are infeasible, as the emulation tools currently require the input structure to be flat — a fixed number of inputs, each with a single value. Therefore, temporal inputs must be modelled as separate inputs for each time period, for example ‘TotalRainMonth01’, ‘TotalRainMonth02’, and so on. If emulation techniques were not applied to AquaCrop, such inputs could be combined into a single input. For example, ‘TotalRain’ would be a collection of total rainfall measurements, each with a specific date. AquaCrop crop characteristics contain several discrete inputs, which are currently unsupported by the emulation tools. To allow the model to be emulated, a duplicate Web service process class was created which does not expose these inputs, and fixes the values to sensible values for wheat, the only crop of interest in the FERA case study.

Data

While an important part of the Model Web, exposing data through Web services is not part of this work, and has been performed by other members of the UncertWeb project. Aiming to provide a high level of interoperability, the majority of model input data will be provided as O&M, hosted

on a Sensor Observation Service (SOS). For uncertain observations, an uncertainty-enabled SOS, an extension to the original specification developed by 52°North, was employed. The use of Web services enables O&M inputs for LCCS and LandSFACTS to be supplied as data references, removing the requirement for the workflow orchestrator to handle this data. All other data, including parameters for AquaCrop, will be specified at either runtime, or workflow composition.

5.3.2 Building the workflow

The workflow required a high level of control and some translation between models, as in some instances it was not possible to directly pass the output from one model to the input of another. While workflow composition and orchestration is not the direct focus of this work, it must be considered when identifying the real challenges in both the Model Web, and uncertainty management in the Model Web.

LCCS outputs uncertain transition matrices, with Dirichlet distributions for each row. As LandSFACTS can only accept transition matrices with probabilities, the distributions must be sampled before LandSFACTS is executed, requiring an additional service to do so. When managing uncertainty in workflows, intermediate sampling is a common requirement, as many models do not inherently support uncertain inputs. A process for sampling uncertain transition matrices, which utilises the matlab-connector library to execute matrix sampling MATLAB code, was developed and exposed on the Web.

The output from LandSFACTS contains simulated crops for each field, for a number of years. In the FERA case study, we are only interested in fields which contain wheat, and thus only want to run AquaCrop for these fields. This requires non-trivial orchestration logic, as the uncertain nature of the workflow means that for a single field, for a given simulation year, there could be $n \times s$ simulated crops, where n is the number of samples drawn from the LCCS uncertain transition matrix, and s is the number of internal LandSFACTS simulations performed. In addition, as LandSFACTS is run n times, there are n separate responses, which must either be combined or checked individually by the orchestrator for occurrences of wheat.

Once the list of fields containing wheat are selected for each year, the orchestrator must retrieve the appropriate weather and soil data from the SOS. This involves a spatial query based on the location of the field, and also a temporal query for the given year. Supplying these inputs to AquaCrop requires a form of translation to be performed, as the weather and soil data are O&M measurements, but as it has no spatial or temporal context, AquaCrop only requires single double

values as inputs. The weather data has already been aggregated to a monthly level, meaning the O&M result value can be directly copied to the relevant AquaCrop input for the month of the measurement.

As there is no spatial or temporal context in AquaCrop, the orchestrator must track individual AquaCrop runs, ensuring that it is aware of which calculated yield output matches which field and year. This burden on the orchestrator could be alleviated if the AquaCrop model accepted O&M inputs. However, this would negatively impact the usability of the model, as a user would be required to create multiple O&M measurements to supply as inputs, and this may seem unnecessary considering the additional metadata provided in the measurements will not actually be used by the model.

The final workflow output is a set of wheat yield estimates grouped by field, for a five year period, and if correctly tracked by the orchestrator, can be assembled as a set of O&M observations. If required, field level yield estimates could be aggregated to regional level, using the UncertWeb developed Spatio-temporal Aggregation Service (STAS). Integration with the Model Web through the adoption of O&M enables other generic tools to be used, such as Greenland⁵, for visualisation of geospatial data. Figure 5.2 shows an example visualisation of the simulated crop yields.

Implementation of the workflow was undertaken by a research assistant on the UncertWeb project. The suitability of BPEL and Taverna to orchestrate the workflow were considered, but excluded in favour of a JavaScript Web client. While it would be possible to create and orchestrate the workflow in Taverna, FERA required spatial visualisations for intermediate results. These visualisations would ideally be provided through a Web browser, requiring complex integration between Taverna, a server-side component, and the Web client. BPEL workflows can be deployed on an engine, after they are available as a Web service, therefore easing the integration problems faced with Taverna. However, BPEL still lacks a truly usable composition client, and problems were encountered adding the required level of workflow control using the Eclipse BPEL Designer.

Exposing the models in the FERA case study using the processing service framework allowed the JavaScript workflow client to be developed with ease. Within the client, requests were built as JavaScript objects and sent to the Web service interface asynchronously using jQuery, a JavaScript library. Upon receiving the response, jQuery automatically parses the response into an object. The adoption of GeoJSON allowed request and response data to be visualised using the OpenLayers mapping library, without additional translation.

⁵<https://wiki.52north.org/bin/view/Geostatistics/Greenland>

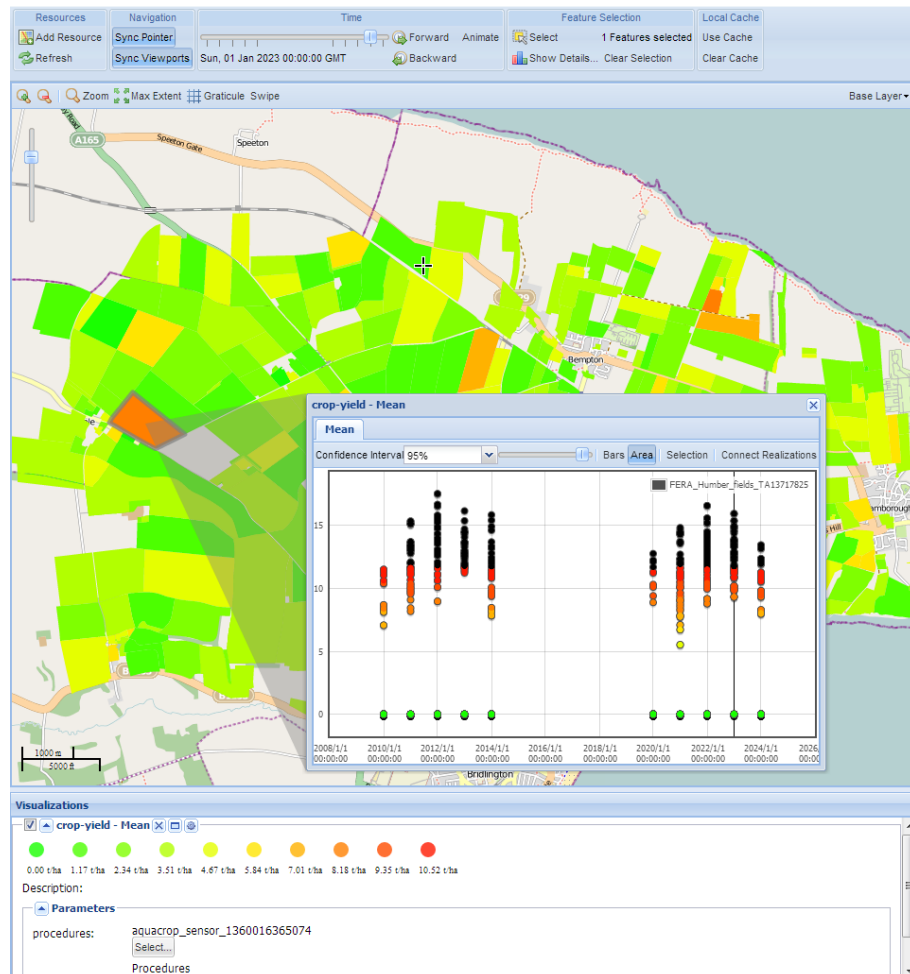


Figure 5.2: Comparing yield estimates over a 24 year period using the Greenland visualisation tool.

Unfortunately, the inclusion of the SOS as a data provider required some of the models to be executed through their SOAP interface, rather than the JSON one. The 52°North SOS implementation does not currently support JSON as a data format, most likely due to the JSON encoding of O&M being non-standardised, thus we must retrieve data as XML from the SOS to pass to the SOAP interface. Although the use of the processing service framework allowed the switch to a SOAP interface to be made without any server-side changes, this demonstrates the challenges faced in the composition of Model Web workflows, and emphasises the need for standardised data formats.

5.4 Emulation of AquaCrop

AquaCrop was identified as a good candidate for emulation, as the model also has no spatial context and accepts only single double values as inputs, and while a single evaluation takes around 10 seconds, it must be evaluated several thousand times in the FERA workflow. The Web service process inputs for the model have already been designed in a manner that enables compatibility with the emulation tools developed in Chapter 4. However, internal conditions within AquaCrop introduced problems when attempting to emulate the model with the tools.

Some AquaCrop inputs are related to other inputs, and their values must satisfy certain conditions. If these conditions are unsatisfied, the model will either fail to run, or produce unrealistic results. For example, each soil horizon must satisfy, $PWP < FC < SAT < kSat$, where PWP , FC and SAT are the soil water content at permanent wilting point, field capacity and saturation respectively, and $kSat$ is saturated hydraulic conductivity. If we create a Latin hypercube sampling (LHS) design for these inputs, there is no guarantee that these conditions will be satisfied, as points are drawn from a range defining whole input space. The same applies for weather data, where we wish to use rainfall, temperature, and evapotranspiration measurements which accurately represent continuous monthly weather data for a year. These input constraints and correlations are likely to occur frequently in modelling scenarios.

Several solutions to the problem were considered, including remodelling the service inputs to enforce the constraints, and creating a language for both the frontend and API to define input constraints. However, although the former option is possible in this case study as the model interfaces can be modified, in other cases this may be impossible. Creating a constraint language is a large amount of research and implementation effort, and was considered outside of the scope of this work. A solution was found in the ability to supply a previously generated design to train

an emulator. This was considered to be the most reusable option, as no changes to existing models are required.

No changes were required to the API, as process evaluation, emulator training and validation are all performed with a user-specified design. Changes were required to the frontend to augment a generated LHS design with points specified explicitly by the user. This allows subsequent frontend requests to the API to send the design as they would previously, without any special treatment required for the user specified points. To enable design import, the simulator specification page was modified. In addition to being able to set a fixed value, or range, a user can also upload CSV file containing design points, referred to as samples. After uploading a CSV file, the user must select which columns in the file contain points for which simulator input (Figure 5.3). If column headings are present, input identifiers are matched automatically with column headings if they share the same value. When augmenting the generated design with user specified points, if the number of specified points is lower than the requested design size, duplicate points from the imported design are included.

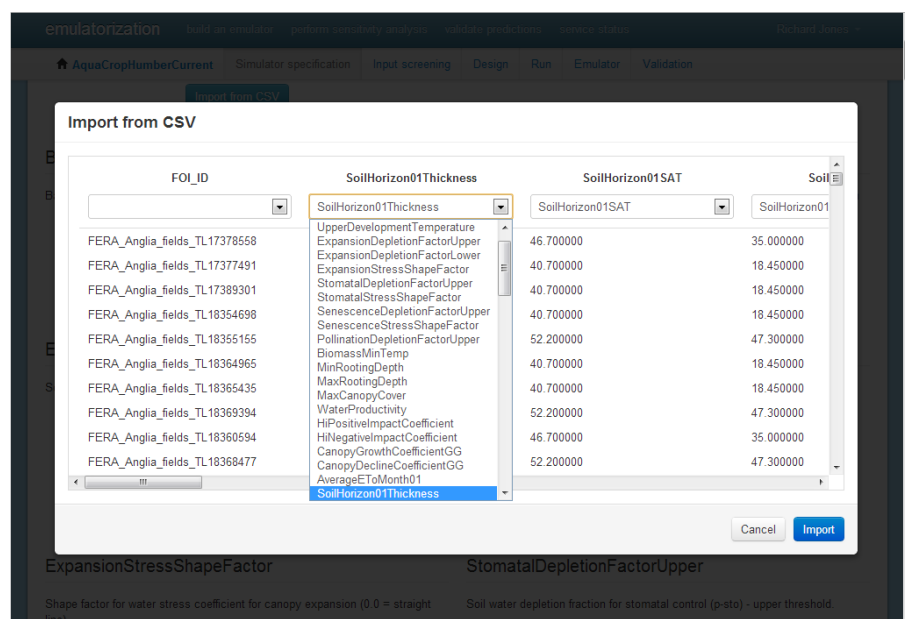


Figure 5.3: Assigning an input identifier to a column of uploaded design points.

Screening and SA methods, including Morris, Sobol, and FAST, implement specific sampling methods to perform their analyses. While it is possible to replace generated design points with those imported by a user, this is likely to produce invalid measures, as the screening or SA method will expect evaluation to have been performed against the generated design. As these sampling methods are range-based, generated designs will rarely satisfy any model input conditions, thus rendering screening and SA unusable if such conditions exist.

The sample import mechanism allowed real soil and simulated climate data to be used to train the emulator, ensuring the input conditions for AquaCrop were satisfied. Both soil data and climate data was from the Yorkshire Wolds NCA, and the climate data consisted of simulations for the current time period. The use of such specific data can help to build a more accurate emulator, as the volume of the input space is reduced. However, this does produce an emulator that may not be suitable for estimating wheat yields outside the Yorkshire Wolds NCA region, or for a different time period.

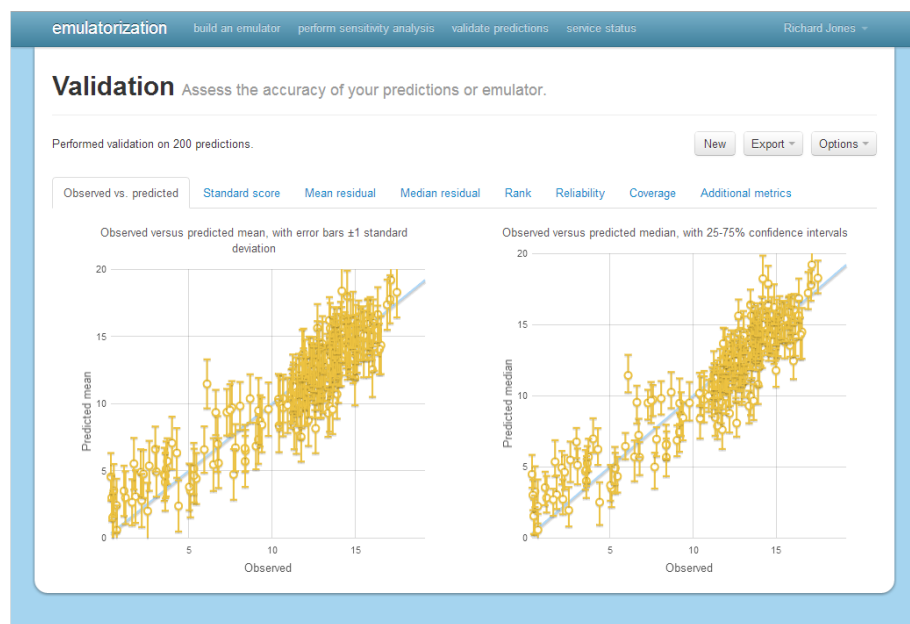


Figure 5.4: AquaCrop simulator versus emulator mean and median plots.

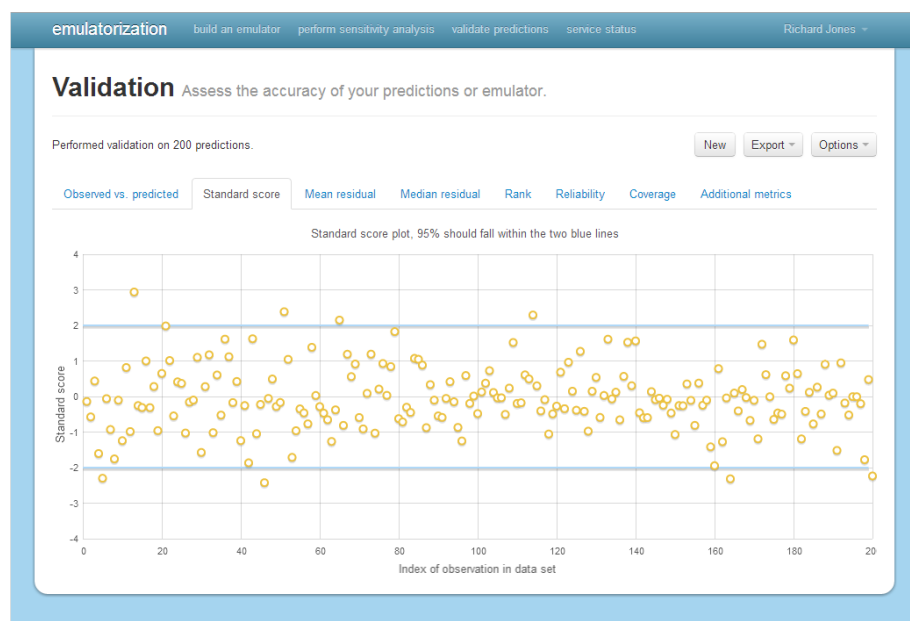


Figure 5.5: Standard score plot for the trained AquaCrop emulator.

Figure 5.4 shows plots comparing simulator (observed) and emulator (predicted) results, for both the mean and median. The majority of points follow the identity line, indicating that emulator predictions are generally accurate. However, there are significant errors on all predictions made by the emulator. A larger design size could potentially reduce prediction errors, but at a cost of having to evaluate the simulator on a greater number of points. Figure 5.5 shows the standard score plot for the AquaCrop emulator. Approximately 95% of scores in the plot are within the two blue lines, and a good spread, suggesting that the emulator has the correct level of confidence when making predictions.

The work in this thesis focuses on the accessibility of emulation, rather than the development and implementation of methods used to build and validate emulators. Therefore, for further details on the parameters used to train the AquaCrop emulator, see Johnson et al. (2012).

The validated emulator was deployed on the emulator upload service, after which it was automatically available through either SOAP/WSDL or JSON based interfaces. Only small changes are required to integrate the emulator with the workflow, including changing the service URL to the emulator location, and modifying the process name to that of the emulator. The request and response handling was modified to reflect the ability of emulators to evaluate multi-point designs. Instead of accepting single values for inputs, the emulator takes arrays, and therefore returns arrays of outputs. This also requires a change to the workflow logic, as the emulator process is only evaluated once, whereas the simulator is evaluated for each design point.

Once the emulator was deployed on the Web, emulator evaluation speeds could be compared fairly with simulator evaluation speeds. A set of designs containing 1, 10, 100, 500, and 1000 points were generated based on the emulator validation data. A simple script was developed to perform repeated simulator evaluations, and a single emulator evaluation against each design, and measure execution times. Table 5.1 shows the results of the emulator and simulator evaluations, clearly demonstrating the large performance gains offered by emulation techniques. While sending multiple requests to the simulator Web service will add transport overheads, the execution of AquaCrop accounts for the vast majority of the evaluation time, which grows linearly with design size. Emulator evaluation time, however, does not grow linearly with design size, and doubling with design size from 500 to 1000 points only increased evaluation time by around 1.5 seconds.

Number of points	Simulator time (s)	Emulator time (s)
1	2.63	1.90
10	26.33	2.14
100	268.69	2.28
500	1326.05	3.53
1000	2652.95	4.96

Table 5.1: A comparison of simulator and emulator evaluation times.

5.5 Summary

The FERA workflow was successfully constructed and orchestrated to produce simulated crop yield estimates for both current and future climate scenarios. However, implementing the case study demonstrates challenges involved in building the Model Web, including exposing existing models on the Web, workflow control and data transformation, and uncertainty management.

Significant implementation effort was required to expose the FERA use case models on the Web, with LandSFACTS and AquaCrop, two third-party compiled language models, being the most challenging. Creating Web service processes for both of these models required understanding of input and output file formats, and mechanisms to generate and parse these automatically. These mechanisms were developed as reusable Java libraries, representing a contribution to the open source modelling community. Problems limiting automated model execution and platform restrictions were overcome by obtaining source code for LandSFACTS, and utilising a compatibility layer and implementing workarounds for AquaCrop. These solutions required a good level of understanding of both computer programming, and the models themselves, and could not be undertaken by non-technical users. Although the use of the processing service framework did remove any requirement to understand Web-based data interchange formats, the objects representing parsed instances of data must still be converted to and from model compatible inputs and outputs.

The Model Web only requires this implementation effort to be performed once, as when a model is exposed on the Web it can be accessed globally in a standard manner, without requiring any knowledge of the underlying implementation. However, enough benefits need to be provided by the Model Web to make the implementation effort worthwhile. In the FERA workflow, these benefits included use of the emulation tools developed in Chapter 4, workflow composition and orchestration, and use of a data visualisation tool.

A workflow for the FERA case study was defined, and a client for orchestration was developed elsewhere in the UncertWeb project. Due to the use of the processing service framework,

the JavaScript client could send JSON encoded requests, and only required use of XML where data services lacked support for JSON. Adopting standard data formats enable the workflow outputs to be visualised in another Model Web tool, the Greenland visualisation client. There were some limitations with the JavaScript client, as when dealing with a large number of observations, Web browsers struggled to cope with handling workflow execution while maintaining responsive. However, these limitations could potentially be removed through code optimisation, using HTML5 Local Storage and Web SQL Database technologies to cache observations which are not required in-memory.

As a candidate for emulation, the AquaCrop Web service process inputs were modelled in a manner that made the service compatible with the emulation tools. Although this does not limit usage of the model for the FERA case study, this choice does remove the ability to supply daily or ten-daily climate measurements for AquaCrop. This also raises a question of the usability of the emulation tools, as if we could not modify the Web service process, the inputs could not be modelled to enable compatibility.

To ensure AquaCrop input conditions were met, the emulation Web frontend was modified to allow import of a previously generated design. Any inputs with an imported design are thus excluded from the existing sampling process, and these uploaded values are used for process evaluation, emulator training, and validation. The validated AquaCrop emulator demonstrated significant performance gains over the simulator, and the emulator upload service allowed the Web-enabled simulator to be switched for the uploaded emulator in the workflow. However, the performance gains come at a cost, and that is the uncertainty introduced by the emulator, which was accounted for in workflow outputs.

Overall, this chapter has tested the applicability of the processing service framework and the emulation tools in a real scenario, thus demonstrating the potential of the Model Web. The framework was employed to expose existing models on the Web, and the emulation tools were used to build and validate an emulator that was subsequently integrated with the case study workflow to provide substantial execution time improvements.

6

Conclusions

CONTENTS

6.1	Thesis summary	158
6.1.1	Development of a framework for exposing models on the Web	158
6.1.2	Development of a Model Web tool for emulation	159
6.1.3	Thesis aims and objectives	160
6.2	Directions for future research	162

6.1 Thesis summary

6.1.1 Development of a framework for exposing models on the Web

Chapter 2 emphasised the current popularity of SOAs, where software components are built as reusable, interoperable services. SOAs underlie the concept of the Model Web — an open, multi-disciplinary collection of interoperable models and data sources, accessible through the Web. When developing SOAs, the choice of service interface is critical to ensure interoperability between services. For the Web, the WPS was most documented for the purpose of exposing geospatial processing functionality, but domain independent SOAP/WSDL interfaces were shown to have good support in client software and libraries.

Chapter 3 evaluated the suitability of the WPS standard for implementing the Model Web. Emphasis was put on usability for end users — those who may not necessarily have a familiarity with Web service and encoding standards, but will benefit extensively from the developments of the Model Web. While the WPS standard provides several uniform features to support the Model Web, and many server-side frameworks are available, several usability problems were identified. These problems are primarily caused by a lack of integration with SOAP and WSDL, resulting in limited support for WPS in client software and libraries. For compatibility with WPS, a tool must either explicitly support the standard, or the WPS user must utilise a proxy service to generate the required WSDL documents.

A solution to the usability issues associated with WPS was provided in the processing service framework. The framework shares some of the uniform features included in the WPS standard, but adopts SOAP and WSDL to provide maximum compatibility with existing tools. As identified in Chapter 2, browser-based applications are becoming increasingly popular, and their reliance on JavaScript provided the motivation to build a JSON-based interface into the framework. From usability tests with a simple, contrived example, the framework was found to be more suitable over WPS for generating client-side code, process execution from JavaScript, and inclusion in Taverna and BPEL workflows.

The effort required to implement Model Web components was strongly emphasised in Chapter 3, which detailed the challenges faced when attempting to expose existing models on the Web. While several strategies were introduced to reduce the implementation effort required to communicate with models written in MATLAB and R, the sheer variation in execution behaviour, data formats and platform compatibility of other models was concluded to be a limiting factor in the

development of the Model Web.

In Chapter 5, the framework was adopted to deploy several models in a case study, with the final aim of constructing a workflow to predict the effects of climate change on wheat yields in England. Several of the difficulties exposing models on the Web identified in Chapter 3 were encountered, as two of the three models were supplied as executable binaries, requiring specific code to enable the Web service interface to execute the model in an automated manner. While this code was developed as open source libraries for each model, meaning another developer would not have to repeat this procedure, implementing model communication was time consuming and not simple. For future model development, the widespread use of a framework such as OpenMI would enable models to be deployed on the Web without any specific code. However, the problem cannot be alleviated for existing models, thus requiring the Model Web to provide significant benefits for the implementation effort to be worthwhile.

The framework proved to be suitable for exposing this set of models on the Web, only requiring a process class to be written for each of the models, leaving the framework to handle data parsing, encoding, reference generating, and error checking. When orchestrating the case study workflow, the framework was advantageous over WPS, as the JSON interface was used to build a JavaScript client. Without JSON, XML parsers would have to be written for each data type, significantly increasing development time.

6.1.2 Development of a Model Web tool for emulation

Chapter 2 discussed various methods for uncertainty management, including UA to quantify the overall output uncertainty as a result of all input uncertainty, and SA study how different sources of input uncertainty contribute to output uncertainty. The application of these methods was found to be limited by the use of multiple model evaluations, which can mean perform a single UA or SA could take several days to complete. This limitation can be removed by the use of emulators, statistical approximations of model simulators which are typically significantly faster to evaluate than a traditional model. However, accessibility of emulator methods is currently poor, requiring the user to undertake several data translation and model evaluation steps manually.

A tool for building emulators, consisting of a backend API and Web frontend, was designed and developed in Chapter 4, solving the problem of accessibility to emulator methods. The backend API enables emulator methods to be integrated with existing software, whilst the frontend supports the entire emulation lifecycle in a user-friendly manner, providing interactive visualisa-

tions and parameter adjustment through forms. Further extensions were made to the tool to provide access to SA and probabilistic validation techniques.

The tool demonstrates the advantages of standard service interfaces and data formats in the Model Web, and may serve to encourage more models to be deployed on the Web. Once a model is exposed on the Web using either the processing service framework or WPS, given the inputs and outputs have data types suitable for emulation, the model will be immediately compatible with the tools. Integration between existing Model Web components and emulators was supported with the development of an extension to the processing service framework, allowing emulators to be uploaded, after which they are available in the same manner as any other model exposed using the framework.

Chapter 5 tested the applicability of the tools in the wheat yield case study, with a specific focus on emulating an existing third-party model for calculating crop yields, AquaCrop. The discovery of input constraints in the AquaCrop model resulted in a design upload extension to the emulation tools, further extending the applicability of the tools to models where range-based sampling may generate invalid input points. An AquaCrop emulator was trained, validated and deployed using the emulator upload service, allowing the simulator in the existing case study workflow to be replaced by the emulator with only a small number of changes. This case study demonstrated the benefits of emulator methods when performing UA, with the emulator evaluating 1000 points in under 5 seconds, compared to over 2500 seconds for multiple simulator evaluations.

While the tools were suitable for building and exposing an emulator for AquaCrop, the complexity of emulator methods, challenges with various Web service data formats and limited development time may restrict usage of the tools to a small number of models. Although emulation may not be applicable in all cases, the work can be considered to be a significant contribution to the Model Web tool set, providing a base for further generic tool developments and encouraging model owners and users to deploy their models on the Web.

6.1.3 Thesis aims and objectives

The conclusions of this thesis are given below, specifically relating the work in each to the objectives defined in Section 1.3.1:

- The emulation backend API and frontend support the construction of emulators, answering the first part of objective 1 outlined in Section 1.3.1, while additionally providing accessibility to SA and probabilistic validation methods. Each step of the emulator building process

is supported by the tool, and the frontend provides a series of parameter adjustment forms and visualisations to help users train and validate an emulator surrogate for a Web-enabled model. The functionality provided by the tool represent a significant contribution to the tool set of the Model Web.

- The processing service framework adopts widely used technologies to expose processes and models on the Web. The framework allows processes to be exposed and consumed using standard interfaces and data encoding formats, without requiring the developer or consumer to have extensive knowledge of these standards. The adoption of the framework in Chapters 3 and 5 concluded that the framework can enhance the interoperability and usability of Web-based models, thus answering objective 2. However, the degree to which this objective has been met cannot be accurately quantified until the framework has been tested for a diverse range of models and use cases.
- A substantial amount of effort is required to deploy existing models on the Web. As the framework simplifies process development, the majority of this effort is in developing mechanisms to communicate with models. To ease this burden in certain cases, a generic connector for MATLAB, and two libraries for third-party models were created, allowing these models, and those written in MATLAB, to be evaluated programmatically from the Web service interface (objective 3).
- An extension to the processing service framework was developed to allow emulators to be uploaded. Once uploaded, an emulator is available through the standard SOAP and JSON interfaces provided by the framework, and can therefore be used with code generation tools, existing clients, and included in a workflow as with any other model, in answer to the second part of objective 1.
- The FERA case study demonstrated that the processing service framework provides a suitable platform for exposing models on the Web, through the exposure of three models to facilitate a wheat yield change prediction workflow. Using the JSON interface, the workflow could be orchestrated in a JavaScript-based Web client, without a need for format-specific XML parsers. The construction and deployment of an emulator for the AquaCrop model is proof that the tools enhance access to emulator methods, and allowed the emulator to be integrated in the existing case study workflow, thus answering both parts of objective 1.

6.2 Directions for future research

The work in this thesis has provided a foundation for uncertainty management in the Model Web. The benefits of utilising the processing service framework and emulation tools have been proved by their use in an uncertainty-enabled workflow, used to predicted the effects of climate change on wheat yields in England. There is significant potential for the processing service framework, emulation tools and the Model Web in general, and some directions these can be taken forward are listed below:

Improve emulation model support — The emulation tools currently only support continuous double encoded inputs and outputs. However, recent work by Tulloch (2013) extended underlying emulation methods by providing support for discrete values, which could be integrated with the backend API and frontend. As discussed in Section 4.5, a wider array of models could be supported if the ability to fix complex input values was provided, thus ignoring them during emulator training. Although this does not allow complex inputs to be considered during emulator training, fixing these values would enable the model to be evaluated because they are required by the model. The ability to emulate models with complex data types, such as O&M, is also a potential avenue for future research, involving a mechanism for specifying which data element contains the input, and how to provide default values for non-required elements, needed to produce valid O&M.

Encourage framework adoption — The processing service framework has been shown to be beneficial when exposing models and geospatial processes, with the support of widely used standards SOAP, WSDL and JSON increasing compatibility with software and tools. However, usability testing has only been performed with a small number of models from a single domain, limiting the potential for a Model Web built using the framework. To see widespread adoption, the framework must be introduced to, and tested in, other domains.

Recommendations for WPS 2.0 — Due to their adoption by the community, standards within the OGC cannot be ignored, with WPS 2.0 currently under development¹. The demonstration of the framework can serve as motivation for the OGC to consider supporting other technologies and methodologies to improve usability of their own standards. For WPS 2.0, recommendations in three critical areas can be made:

¹<http://www.opengeospatial.org/projects/groups/wps2.0swg>

Technology integration — Client and workflow support for WPS processes is currently limited, often requiring non-standard extensions and proxy services. By integrating with widely used technologies, including the support of process-specific WSDL documents and JSON requests, the effort required by the process consumer is minimised.

Profile support — While application profiles provide common descriptions for processes which have the same inputs and outputs, in practice this mechanism is only suitable for exposing multiple implementations of a single algorithm. With the creation of domain-specific profiles, inputs and outputs could instead be restricted to a set of data types commonly used in the target domain, enabling clients to guarantee support for any process implementing that profile.

Message descriptions — As complex WPS process inputs and outputs are described by referencing a schema URL, restricting data types to individual elements is only possible through the creation of separate schemas. This approach is time consuming, and may break clients which rely on schema URLs to be well-known. Instead, it should be possible for input and output descriptions to reference both a schema URL and allowed elements from that schema.

Ease model exposure — The work in this thesis included a discussion on the strategies for automated model communication from a Web service interface, and included the development of libraries to communicate with two existing models and a generic connector for those written in MATLAB. However, exposing a model on the Web still requires significant implementation effort, thus limiting the adoption of the Model Web. If models share common characteristics, or describe themselves in a standard way, it is possible to automate the process of exposing an model on the Web. While such mechanisms have been developed for R and WPS integration, MATLAB model functions could also be annotated to describe their input and output parameters, allowing a tool to automatically generate and deploy a Web service process to communicate with that model. Gupta et al. (2012) demonstrated the ability to create a generic Web service process for OpenMI models, but further research is necessary to ensure compatibility with time-stepping models and the latest version of the interface.

Ease data exposure — While models are the focus for the work in this thesis, data sources are a significant component in the Model Web. Standards exist for exposing geospatial data, such

as WFS and SOS, but as with models, it can be extremely challenging to deploy data on these services, even for those familiar with the standards. To complement those for exposing models on the Web, tools could be developed to ease the effort required to expose data. For example, a user could import data in CSV format, with a GUI-based tool allowing the user to select which column contains the result, which column contains the feature coordinates, and so on.

Improve workflow software — The processing service framework supports popular standards, aiming to make process consumption possible in a variety of software and tools. However, for composing Model Web workflows, software such as Taverna and the Eclipse BPEL Designer are still far from providing a user-friendly experience. Through the use of standard interfaces and data encoding formats in the Model Web, these tools could be extended to provide a true drag and drop experience. If we consider the FERA use case, the uncertain transition matrix output returned from the LCCS model, after being sampled, is an input to the LandSFACTS. Ideally, it would be possible to drag and connector from the LCCS output to the LandSFACTS input, and have the workflow software manage the intermediate sampling process and output/input mapping.

These contributions and directions for future research open up interesting possibilities for the development of tools to support uncertainty management and mechanisms for exposing models in a standard manner, and represent a basis for the Model Web.

Bibliography

- Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Sheth, A., and Verma, K. Web Service Semantics - WSDL-S. W3C Note, World Wide Web Consortium (W3C), April 2005. URL <http://www.w3.org/2005/04/FSWS/Submissions/17/WSDL-S.htm>.
- Bastin, L. and Williams, M. UncertML requirements. Deliverable, UncertWeb, November 2010. URL <http://www.uncertweb.org/documents/deliverables>.
- Bastin, L., Cornford, D., Jones, R., Heuvelink, G. B., Pebesma, E., Stasch, C., Nativi, S., Mazzetti, P., and Williams, M. Managing uncertainty in integrated environmental modelling: The UncertWeb framework. *Environmental Modelling & Software*, 39(0):116 – 134, 2013. ISSN 1364-8152. URL <http://www.sciencedirect.com/science/article/pii/S1364815212000564>.
- Bastos, L. and O'Hagan, A. Diagnostics for gaussian process emulators. *Technometrics*, 51(4): 425–438, 2009.
- Benson, E., Wasson, G., and Humphrey, M. Evaluation of UDDI as a Provider of Resource Discovery Services for OGSA-based Grids. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 9–pp. IEEE, 2006.
- Bieberstein, N., Bose, S., Walker, L., and Lynch, A. Impact of service-oriented architecture on enterprise systems, organizational structures, and individuals. *IBM systems journal*, 44(4):691–708, 2005.
- Boukouvalas, A., Cornford, D., and Singer, A. Managing uncertainty in complex stochastic models: design and emulation of a Rabies model. In *6th St. Petersburg Workshop on Simulation*, pages 839–841, 2009.
- Box, D., Ehnebuske, D., Kakivayam, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte,

- S., and Winer, D. Simple Object Access Protocol (SOAP) 1.1. W3C Note, World Wide Web Consortium (W3C), May 2000. URL <http://www.w3.org/TR/soap11>.
- Brauner, J. and Schäffer, B. Integration of GRASS functionality in web based SDI service chains. In *Proceedings of the FOSS4G 2008*, 2008.
- Brauner, J., Foerster, T., Schaeffer, B., and Baranski, B. Towards a research agenda for geoprocessing services. In *12th AGILE International Conference on Geographic Information Science*. Hanover, Germany: IKG, Leibniz University of Hanover, 2009.
- Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, World Wide Web Consortium (W3C), November 2008. URL <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- Campolongo, F., Cariboni, J., and Saltelli, A. An effective screening design for sensitivity analysis of large models. *Environmental Modelling & Software*, 22(10):1509 – 1518, 2007. ISSN 1364-8152. URL <http://www.sciencedirect.com/science/article/pii/S1364815206002805>.
- Chan, K., Saltelli, A., and Tarantola, S. Sensitivity analysis of model output: variance-based methods make the difference. In *Proceedings of the 29th conference on Winter simulation*, pages 261–268. IEEE Computer Society, 1997.
- Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. Web Services Description Language (WSDL) 1.1. W3C Note, World Wide Web Consortium (W3C), March 2001. URL <http://www.w3.org/TR/wsdl>.
- Christensen, F. A., Bernard, L., Kanellopoulos, I., Iso, N. J., Peedell, S., Schade, S., and Thorne, C. Building service oriented applications on top of a spatial data infrastructure - a forest fire assessment example. In *Proceedings of the 9th International AGILE Conference*, April 2006.
- Conti, S. and O'Hagan, A. Bayesian emulation of complex multi-output and dynamic computer models. *Journal of statistical planning and inference*, 140(3):640–651, 2010.
- Cornford, D. and Williams, M. UncertML best practice proposal. Deliverable, UncertWeb, March 2011. URL <http://www.uncertweb.org/documents/deliverables>.
- Crasso, M., Rodriguez, J. M., Zunino, A., and Campo, M. Revising WSDL Documents: Why and How. *IEEE Internet Computing*, 14:48–56, 2010. ISSN 1089-7801.

- Cukier, R., Fortuin, C., Shuler, K., Petschek, A., and Schaibly, J. Study of the sensitivity of coupled reaction systems to uncertainties in rate coefficients. I Theory. *The Journal of Chemical Physics*, 59(8):3873, 1973.
- Currin, C., Mitchell, T., Morris, M., and Ylvisaker, D. Bayesian Prediction of Deterministic Functions, with Applications to the Design and Analysis of Computer Experiments. *Journal of the American Statistical Association*, 86(416):953–963, 1991. URL <http://amstat.tandfonline.com/doi/abs/10.1080/01621459.1991.10475138>.
- Czitrom, V. One-Factor-at-a-Time Versus Designed Experiments. *The American Statistician*, 53(2):126–131, 1999.
- de Jesus, J., Walker, P., Grant, M., and Groom, S. WPS orchestration using the Taverna workbench: The eScience approach. *Computers & Geosciences*, 47:75–86, 2011.
- Dubois, G., Skøien, J., Peedell, S., De Jesus, J., Geller, G., and Hartley, A. eHabitat: a contribution to the model web for habitat assessments and ecological forecasting. In *Proceedings of the 34th International Symposium on Remote Sensing of Environment. April*, pages 10–15, 2011.
- Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. Technical report, The Internet Society, June 1999.
- Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- Foerster, T., Bruehl, A., and Schaeffer, B. RESTful Web Processing Service. In *Proceedings of the 14th AGILE International Conference on Geographic Information Science*, 2011.
- Frey, H. and Patil, S. Identification and review of sensitivity analysis methods. *Risk analysis*, 22(3):553–578, 2002.
- Geller, G. and Turner, W. The model web: a concept for ecological forecasting. In *Geoscience and Remote Sensing Symposium, 2007. IGARSS 2007. IEEE International*, pages 2469–2472, july 2007.
- Gerharz, L., Stasch, C., Pross, B., Pebesma, E., Jones, R., Williams, M., and Cornford, D. Report on simple and publicly accessible UncertWeb examples. Deliverable, UncertWeb, September 2011. URL <http://www.uncertweb.org/documents/deliverables>.

- Goldstein, M. and Rougier, J. Reified Bayesian modelling and inference for physical systems. *Journal of Statistical Planning and Inference*, 139(3):1221–1239, 2009.
- Gregersen, J., Gijssbers, P., and Westen, S. OpenMI: Open modelling interface. *Journal of Hydroinformatics*, 9(3):175–191, 2007.
- Gupta, T., Jones, R., Bastin, L., and Cornford, D. Integrating OpenMI and UncertWeb: Managing Uncertainty in OpenMI models. In *2012 International Congress on Environmental Modelling and Software*, pages 1151–1158, 2012.
- Hamby, D. M. A review of techniques for parameter sensitivity analysis of environmental models. *Environmental Monitoring and Assessment*, 32:135–154, 1994. ISSN 0167-6369. URL <http://dx.doi.org/10.1007/BF00547132>. 10.1007/BF00547132.
- Heuvelink, G. *Error propagation in environmental modelling with GIS*. CRC Press, 1998.
- Hollingsworth, D. The Workflow Reference Model. The workflow management coalition specification, Workflow Management Coalition, 1995.
- Homma, T. and Saltelli, A. Importance measures in global sensitivity analysis of nonlinear models. *Reliability Engineering & System Safety*, 52(1):1–17, 1996.
- Iman, R. L. and Hora, S. C. A Robust Measure of Uncertainty Importance for Use in Fault Tree System Analysis. *Risk Analysis*, 10(3):401–406, 1990. ISSN 1539-6924. URL <http://dx.doi.org/10.1111/j.1539-6924.1990.tb00523.x>.
- Johnson, J., Gosling, J.-P., and Cornford, D. Requirements for service chains in the land-use application domain. Deliverable, UncertWeb, July 2010. URL <http://www.uncertweb.org/documents/deliverables>.
- Johnson, J., Knight, S., Cornford, D., and Jones, R. An UncertWeb service for land-use change modelling. Deliverable, UncertWeb, November 2012. URL <http://www.uncertweb.org/documents/deliverables>.
- Jones, R., Bastin, L., Cornford, D., and Williams, M. Handling and communicating uncertainty in chained geospatial web services. In *Proceedings of the Ninth International Symposium on Spatial Accuracy Assessment in Natural Resources and Environmental Sciences*, 2010.

- Jones, R., Cornford, D., and Bastin, L. UncertWeb Processing Service: Making Models Easier to Access on the Web. *Transactions in GIS*, 16(6):921–939, 2012. ISSN 1467-9671. URL <http://dx.doi.org/10.1111/j.1467-9671.2012.01328.x>.
- Juric, M., Rozman, I., Brumen, B., Colnaric, M., and Hericko, M. Comparison of performance of Web services, WS-Security, RMI, and RMI–SSL. *Journal of Systems and Software*, 79(5): 689–700, 2006.
- Kennedy, M. C. and O’Hagan, A. Bayesian calibration of computer models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(3):425–464, 2001.
- Kennedy, M., Anderson, C., Conti, S., and O’Hagan, A. Case studies in Gaussian process modelling of computer codes. *Reliability Engineering & System Safety*, 91(10):1301–1309, 2006.
- Krzykacz, B. *Samos: a computer program for the derivation of empirical sensitivity measures of results from large computer models*. Ges. für Reaktorsicherheit (GRS), Forschungsgelände, 1990.
- Lake, R. Profiles Make GML Easier to Swallow. *GEO WORLD*, 18(9):24, 2005.
- Lim, B. and Wen, H. J. Web Services: An Analysis of the Technology, its Benefits, and Implementation Difficulties. *Information Systems Management*, 20(2):49–57, 2003. URL <http://www.tandfonline.com/doi/abs/10.1201/1078/43204.20.2.20030301/41470.8>.
- Lopez-Pellicer, F. J., Rentería-Agualimpia, W., Béjar, R., Muro-Medrano, P. R., and Zarazaga-Soria, F. J. Availability of the OGC geoprocessing standard: March 2011 reality check. *Computers & Geosciences*, 47:13 – 19, 2012. ISSN 0098-3004. URL <http://www.sciencedirect.com/science/article/pii/S009830041100358X>.
- Louridas, P. Orchestrating Web Services with BPEL. *IEEE Software*, 25:85–87, March 2008. ISSN 0740-7459. URL <http://dl.acm.org/citation.cfm?id=1345866.1345902>.
- Mazzetti, P., Nativi, S., and Caron, J. RESTful implementation of geospatial services for Earth and Space Science applications. *International Journal of Digital Earth*, 2(1 supp 1):40–61, 2009. URL <http://dx.doi.org/10.1080/17538940902866153>.
- McKay, M. *Evaluating prediction uncertainty*. The Commission, 1995.

- Meng, X., Bian, F., and Xie, Y. Research and Realization of Geospatial Information Service Orchestration Based on BPEL. *Environmental Science and Information Application Technology, International Conference on*, 2:642–645, 2009.
- Michaelis, C. and Ames, D. Evaluation and implementation of the OGC web processing service for use in client-side GIS. *GeoInformatica*, 13(1):109–120, 2009.
- Morris, M. D. Factorial Sampling Plans for Preliminary Computational Experiments. *Technometrics*, 33(2):161–174, 1991.
- Nash, E. WPS application profiles for generic and specialised processes. *Geographic Information Days*, 32:69–79, 2008.
- Nativi, S., Mazzetti, P., and Geller, G. Environmental model access and interoperability: The GEO Model Web initiative. *Environmental Modelling & Software*, 2012.
- Newmarch, J. A Critique of Web Services. In *IADIS International Conference e-Commerce 2004*, pages 391–398. IADIS, 2004. URL <http://www.iadisportal.org/digital-library/a-critique-of-web-services>.
- Nüst, D. and Pebesma, E. R in the Sensor Web. In *Sensing a Changing World 2*, 2012.
- Oakley, J. and O’Hagan, A. Bayesian inference for the uncertainty distribution of computer model outputs. *Biometrika*, 89(4):769–784, 2002. URL <http://biomet.oxfordjournals.org/content/89/4/769.abstract>.
- Oakley, J. E. and O’Hagan, A. Probabilistic sensitivity analysis of complex models: a Bayesian approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 66(3): 751–769, 2004. ISSN 1467-9868. URL <http://dx.doi.org/10.1111/j.1467-9868.2004.05304.x>.
- OASIS, . Web Services Business Process Execution Language Version 2.0. OASIS Standard, OASIS, April 2007. URL <http://www.w3.org/TR/soap11>.
- OGC 05-007r7, . *OpenGIS Web Processing Service*. OpenGIS Standard. Open Geospatial Consortium Inc., Wayland, USA, 2007.
- OGC 06-121r9, . *OGC Web Services Common Standard*. OGC Implementation Standard. Open Geospatial Consortium Inc., Wayland, USA, 2010.

- OGC 07-006r1, . *OpenGIS Catalogue Services Specification*. OpenGIS Standard. Open Geospatial Consortium Inc., Wayland, USA, 2007.
- OGC 07-022r1, . *Observations and Measurements – Part 1 – Observation schema*. OGC Standard. Open Geospatial Consortium Inc., Wayland, USA, December 2007.
- OGC 07-036, . *OpenGIS Geography Markup Language (GML) encoding standard: version 3.2.1*. OGC Standard. Open Geospatial Consortium Inc., Wayland, USA, 2007.
- OGC 07-041r1, . *CUAHSI WaterML*. OGC Discussion Paper. Open Geospatial Consortium Inc., Wayland, USA, May 2007.
- OGC 07-057r7, . *OpenGIS Web Map Tile Service Implementation Standard*. OpenGIS Standard. Open Geospatial Consortium Inc., Wayland, USA, 2010.
- OGC 08-009r1, . *OWS 5 SOAP/WSDL Common Engineering Report*. OGC Discussion Paper. Open Geospatial Consortium Inc., Wayland, USA, 2008.
- OGC 08-122r2, . *Uncertainty Markup Language (UnCertML)*. OpenGIS Discussion Paper. Open Geospatial Consortium Inc., Wayland, USA, 2007.
- O'Hagan, A. Bayesian analysis of computer code outputs: A tutorial. *Reliability Engineering & System Safety*, 91(10-11):1290–1300, 2006. ISSN 0951-8320. URL <http://www.sciencedirect.com/science/article/pii/S0951832005002383>.
- O'Hagan, A. Probabilistic uncertainty specification: Overview, elaboration techniques and their application to a mechanistic model of carbon flux. *Environmental Modelling & Software*, 2011.
- O'Hagan, A. and Kingman, J. F. C. Curve Fitting and Optimal Design for Prediction. *Journal of the Royal Statistical Society. Series B (Methodological)*, 40(1):pp. 1–42, 1978. ISSN 00359246. URL <http://www.jstor.org/stable/2984861>.
- Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M., Wipat, A., and others, . Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
- Padberg, A. and Greve, K. Gridification of OGC web services: Challenges and potential. *GIS Science* 14, 3:77–81, 2009. URL <http://www.scopus.com/inward/record.url?eid=2-s2.0-77952984286&partnerID=40&md5=982ebdaa8da4cc4b2f78be51f0e7a676>.

- Pannell, D. Sensitivity analysis of normative economic models: theoretical framework and practical strategies. *Agricultural economics*, 16(2):139–152, 1997.
- Papajorgji, P., Beck, H., and Braga, J. An architecture for developing service-oriented and component-based environmental models. *Ecological Modelling*, 179(1):61–76, 2004.
- Papazoglou, M. Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12, December 2003.
- Papoulis, A. *Probability, Random Variables, and Stochastic Processes*. Mc-Graw Hill, 1984.
- Pautasso, C. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.
- Pautasso, C., Zimmermann, O., and Leymann, F. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *17th International World Wide Web Conference (WWW2008)*, pages 805–814, Beijing, China, April 2008. URL <http://www2008.org/>.
- Percivall, G. ISO 19119 and OGC Service architecture. In *FIG XXII International Congress*, April 2002.
- Raape, U., Teßmann, S., Wytzisk, A., Steinmetz, T., Wnuk, M., Hunold, M., Strobl, C., Stasch, C., Walkowski, A., Meyer, O., and others, . Decision support for tsunami early warning in Indonesia: The Role of OGC Standards. *Geographic Information and Cartography for Risk and Crisis Management*, pages 233–247, 2010.
- Rabitz, H. Systems Analysis at the Molecular Scale. *Science*, 246(4927):221–226, 1989. URL <http://www.sciencemag.org/content/246/4927/221.abstract>.
- Refsgaard, J., Van der Sluijs, J., Brown, J., and Van der Keur, P. A framework for dealing with uncertainty due to model structure error. *Advances in Water Resources*, 29(11):1586–1597, 2006.
- Rougier, J. Efficient emulators for multivariate deterministic functions. *Journal of Computational and Graphical Statistics*, 17(4):827–843, 2008.
- Sacks, J., Welch, W. J., Mitchell, T. J., and Wynn, H. P. Design and Analysis of Computer Experiments. *Statistical Science*, 4(4):409–423, 1989. ISSN 08834237. URL <http://dx.doi.org/10.2307/2245858>.

- Saltelli, A., Tarantola, S., and Chan, K. P.-S. A Quantitative Model-Independent Method for Global Sensitivity Analysis of Model Output. *Technometrics*, 41(1):39–56, 1999. URL <http://www.tandfonline.com/doi/abs/10.1080/00401706.1999.10485594>.
- Saltelli, A., Chan, K., and Scott, E. *Sensitivity analysis*, volume 134. Wiley New York, 2000.
- Saltelli, A., Tarantola, S., Campolongo, F., and Ratto, M. *Sensitivity analysis in practice: a guide to assessing scientific models*. Wiley, 2004.
- Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., and Tarantola, S. *Global sensitivity analysis: the primer*. Wiley-Interscience, 2008.
- Saltelli, A. and Bolado, R. An alternative way to compute Fourier amplitude sensitivity test (FAST). *Computational Statistics & Data Analysis*, 26(4):445 – 460, 1998. ISSN 0167-9473. URL <http://www.sciencedirect.com/science/article/pii/S0167947397000431>.
- Sancho-Jiménez, G., Béjar, R., Latre, M., and Muro-Medrano, P. A method to derivate SOAP interfaces and WSDL metadata from the OGC web processing service mandatory interfaces. *Advances in Conceptual Modeling—Challenges and Opportunities*, pages 375–384, 2008.
- Santner, T., Williams, B., and Notz, W. *The design and analysis of computer experiments*. Springer, 2003.
- Schäffer, B. and Foerster, T. A client for distributed geo-processing and workflow design. *Journal of Location Based Services*, 2(3):194–210, 2008. URL <http://www.tandfonline.com/doi/abs/10.1080/17489720802558491>.
- Shade, S., Ostländer, N., Granell, C., Shulz, M., McInerney, D., Dubois, G., Vaccari, L., Chinosi, M., Díaz, L., Bastin, L., and others, . Which Service Interfaces fit the Model Web? In *GEO-Processing 2012, The Fourth International Conference on Advanced Geographic Information Systems, Applications, and Services*, pages 1–6, 2012.
- Shahsavani, D. and Grimvall, A. Variance-based sensitivity analysis of model outputs using surrogate models. *Environmental Modelling & Software*, 26(6):723–730, 2011.
- Skøien, J., Schulz, M., Dubois, G., Jones, R., Heuvelink, G., and Cornford, D. Uncertainty propagation in chained web based modeling services: the case of eHabitat. In *Innovation in*

- sharing environmental observations and information. Proceedings of EnviroInfo 2011, 25th International Conference Environmental Informatics*, pages 46–58, 2011.
- Sobol, I. Sensitivity analysis for non-linear mathematical models. *Mathematical Modeling & Computational Experiment (English Translation)*, 1:407–414, 1993.
- Staab, S., van der Aalst, W., Benjamins, V. R., Sheth, A., Miller, J. A., Bussler, C., Maedche, A., Fensel, D., and Gannon, D. Web Services: Been There, Done That? *IEEE Intelligent Systems*, 18:72–85, 2003. ISSN 1541-1672.
- Stollberg, B. and Zipf, A. Geoprocessing services for spatial decision support in the domain of housing market analyses. In *Proceedings AGILE*, 2008a.
- Stollberg, B. and Zipf, A. OGC Web Processing Service Interface for Web Service Orchestration Aggregating Geo-processing Services in a Bomb Threat Scenario. In Ware, J. and Taylor, G., editors, *Web and Wireless Geographical Information Systems*, volume 4857 of *Lecture Notes in Computer Science*, pages 239–251. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76923-1. URL http://dx.doi.org/10.1007/978-3-540-76925-5_18. 10.1007/978-3-540-76925-5_18.
- Stollberg, B. and Zipf, A. Development of a WPS Process Chaining Tool and Application in a Disaster Management Use Case for Urban Areas. In *Proceedings of the 27th Urban Data Management Symposium*, 2008b.
- Tan, W., Missier, P., Madduri, R., and Foster, I. Building Scientific Workflow with Taverna and BPEL: A Comparative Study in caGrid. In Feuerlicht, G. and Lamersdorf, W., editors, *Service-Oriented Computing — ICSOC 2008 Workshops*, pages 118–129. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-01246-4. URL http://dx.doi.org/10.1007/978-3-642-01247-1_11.
- Tulloch, S. Gaussian Processes with Continuous and Categorical Inputs. Master’s thesis, Aston University, 2013.
- Turányi, T. Sensitivity analysis of complex kinetic systems. tools and applications. *Journal of Mathematical Chemistry*, 5(3):203–248, 1990.
- Yu, G., Zhao, P., Di, L., Chen, A., Deng, M., and Bai, Y. BPELPower-A BPEL execution engine for geospatial web services. *Computers & Geosciences*, 2012.

Zhang, D., Yu, L., Xie, B., and Di, L. Open Geospatial Information Services Chaining Based on OGC Specifications and Processing Model. In *Proceedings of the 2008 International Workshop on Education Technology and Training & 2008 International Workshop on Geoscience and Remote Sensing - Volume 02*, ETTANDGRS '08, pages 153–157, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3563-0. URL <http://dx.doi.org/10.1109/ETTandGRS.2008.63>.

Zhao, P., Foerster, T., and Yue, P. The Geoprocessing Web. *Computers & Geosciences*, 47(0): 3–12, 2012. ISSN 0098-3004. URL <http://www.sciencedirect.com/science/article/pii/S0098300412001446>.